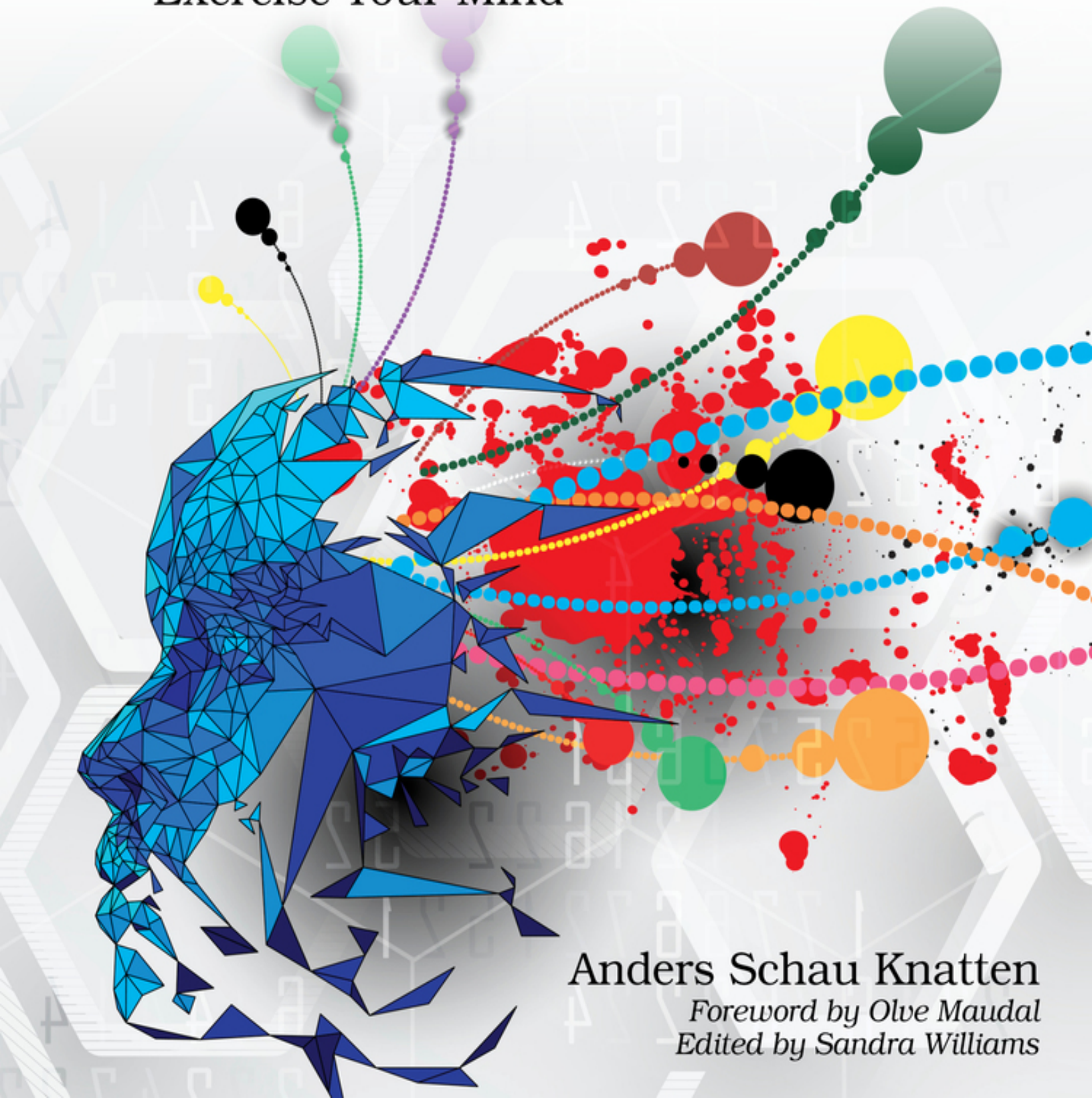
The Pragmatic Programmers

C++ Brain Teasers

Exercise Your Mind



Anders Schau Knatten

Foreword by Olve Maudal

Edited by Sandra Williams

C++ Brain Teasers

Exercise Your Mind

by Anders Schau Knatten

Version: P1.0 (June 2024)

Copyright © 2024 The Pragmatic Programmers, LLC. This book is licensed to the individual who purchased it. We don't copy-protect it because that would limit your ability to use it for your own purposes. Please don't break this trust—you can use this across all of your devices but please do not share this copy with other members of your team, with friends, or via file sharing services. Thanks.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We’re here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://pragprog.com.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it’s still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We’re here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's polices. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/#about-ebooks. To learn more about this book and access the free resources, go to https://pragprog.com/book/akbrain, the book's homepage.

Thanks for your continued support,

The Pragmatic Bookshelf

The team that produced this book includes: Dave Thomas (Publisher), Janet Furlow (COO), Susannah Davidson (Executive Editor), Sandra Williams (Development Editor), L. Sakhi MacMillan (Copy Editor), Potomac Indexing, LLC (Indexing), Gilson Graphics (Layout)

For customer support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Table of Contents

1. **Foreword by Olve Maudal**
2. **Acknowledgments**
3. **Preface**
    1. How to Use This Book
    2. Undefined Behavior
    3. Unspecified and Implementation-Defined Behavior
    4. Playing Around with the Code

---

4. **Part I. C++ Brain Teasers**

Puzzle

1. **1. How Many Degrees?**

Puzzle

2. **2. String Theory**

Puz

zle

3. **3.** **Hack the Planet!**

Puz

zle

4. **4.** **Going Global**

Puz

zle

5. **5.** **A Destructive Relationship**

Puz

zle

6. **6.** **Who’s Up First?**

Puz

zle

7. **7.** **All Good Things Must Come to an End**

Puz

zle

8. **8.** **Will It Move?**

Puzzle

9. **9. Counting Copies**

Puzzle

10. **10. A Strange Assignment**

Puzzle

11. **11. What's the Time of Death?**

Puzzle

12. **12. A False Start**

Puzzle

13. **13. A Constant Struggle**

Puzzle

14. **14. Aristotle's Sum of Parts**

Puz

zle

15. **15. Back from the Future**

Puz

zle

16. **16. An Overloaded Container**

Puz

zle

17. **17. A Strong Point**

Puz

zle

18. **18. Moving Out**

Puz

zle

19. **19. A Little Sum Thing**

Puz

zle

20. **20. Monsters on the Move**

Puz
zle
21. 21. Sizing Up Some Characters

Puz
zle
22. 22. The Phantom Spaceship

Puz
zle
23. 23. Off to a Good Start

Puz
zle
24. 24. A Specialized String Theory

Puz
zle
25. 25. Weakly Typed, Strongly Puzzling

Copyright © 2024, The Pragmatic Bookshelf.

Early Praise for *C++ Brain Teasers*

If you are either a C++ enthusiast or a professional C++ developer—hopefully both—this book is for you. Twenty-five puzzles explain in easy C++ the sometimes-surprising behavior of the language and what you might want to know to steer clear of potential pitfalls and disappointments.

→ Daniela Engert
ISO C++ Committee Member and Keynote Speaker

The inexperienced programmer will learn, in a lighthearted way, of many pitfalls in the C++ programming language and how to avoid them. The seasoned veteran, who has developed an intuition for what to avoid, will learn in great detail how and why the language works the way it does.

→ Björn Fahller
International C++ Conference Speaker

This is a fun and engaging puzzle book for everyone who wants to test their C++ knowledge. The puzzles are fun to solve, and the explanations are easy to follow.

→ Tina Ulbrich

International C++ Conference Speaker

Foreword by Olve Maudal

I first met Anders Schau Knatten at local C++ meetups here in Oslo way back. Since then, we have been friends hanging out when we meet at conferences. Anders has given presentations at C++ focused conferences such as ACCU, NDC TechTown, CppCon, C++ on Sea, and Meeting C++, to mention a few. When I attend one of his talks, I always learn something new. Anders can speak for hours about how integers really behave, or about all the stuff that happens before main() is invoked, or how to use debuggers effectively. He also has this fascinating ability to observe something, adapt it, reshape it, and then take it to the next level. This was exactly what he did after attending a couple of my infamous C++ Pub Quiz sessions. Anders soon went off to create the wonderful site cppquiz.org[1] (try it!). And now he is running his own much-improved version of C++ Pub Quiz sessions at conferences. With this book, Anders is taking his own concept even further.

I love books, I love C++, and I love puzzles. *C++ Brain Teasers* brings all three together. I have been using and studying C++ for more than 30 years now, but I'm sure I will read this book again and again and still learn something new each time. If you are deeply invested in C++, you'll want to read it too. The book serves as a fast-track into a deeper understanding of this extremely powerful and fascinating programming language. The book

contains 25 carefully selected C++ puzzles that will challenge your current understanding of the language. Once you have attempted to solve a puzzle yourself, Anders will give you a deep and solid explanation as to why the puzzle is interesting and how you can learn to reason about these things.

You can write beautiful programs in C++, but it is of course also possible to mess it up completely. It is important to understand that C++ is what it is because it evolved from BCPL, C, Simula 67, and Algol 68. C++ is what it is because the language was adopted fast and already had millions of active users during the early formation years. C++ is what it is because it trusts the programmer and focuses on solving real problems. Yes, C++ is complex, perhaps too complex, but it is indeed the go-to language (pun intended!) for many domains, such as operating systems, embedded systems, high-performance computing, performance-critical applications, video games, and space probes. Indeed, here is a very insightful quote by Bjarne Stroustrup (the creator of C++):

> *There are only two kinds of languages: the ones people complain about and the ones nobody uses.*

Wishing you all the best on your journey toward a deeper understanding of C++. Anders will be your guide this time. Enjoy the ride!

Olve Maudal

International C and C++ Speaker, Trainer, and Conference Organizer

June, 2024

FOOTNOTES

[1] https://cppquiz.org

Copyright © 2024, The Pragmatic Bookshelf.

Acknowledgments

First, I want to thank Olve Maudal and the ACCU, who inspired me to create CppQuiz.org[2] at the ACCU 2013 conference. I would also like to thank everyone who has contributed to CppQuiz.org with content, implementation, and bug reports over the years.

Big thanks to my editor, Sandra Williams, for vital help along the way. Without your guidance, support, and advice, this book would have been much poorer. Also, thanks to Margaret Eldridge for reaching out on behalf of the publisher and Frances Buontempo for introducing us.

I am fortunate to have had help from some of the best C++ people I know. Thank you Daniela Engert, Björn Fahller, Olve Maudal, Karthik Nishanth, Tom Schultz, Tina Ulbrich, Sergei Vasilchenko, and Piotr Wierciński for your help with technical review!

Finally, I want to thank my wife, Charlotte, whose support has allowed me to dedicate the time and effort needed to write this book, and Kristiane and Sebastian for their encouragement.

FOOTNOTES

[2] https://cppquiz.org

Copyright © 2024, The Pragmatic Bookshelf.

Preface

C++ is one of the largest and oldest programming languages in common use. It's famous for getting all the default behaviors wrong and for, as is the famous example in the C++ community, making demons fly out of your nose.[3] You couldn't have picked a better language for a programming brain teasers book!

Through 25 puzzles we'll explore how C++ works under the hood, including some of its important quirks. To get the most out of the book, you should have some experience writing C++ and be familiar with the basics of the language, including simple object-oriented programming and templates.

After finishing the book, you'll have a deeper understanding of topics like initialization, lifetimes, overload resolution, implicit conversions, inheritance, undefined behavior, and more. But more importantly, I hope you'll have gained a sense of curiosity about how C++ really works, even if only a few of those answers fit in a single book.

How to Use This Book

The book contains 25 C++ puzzles with answers and explanations. Most will be well-formed programs, which the C++ standard guarantees the output of. Some might, however, have a compilation error, and some might have undefined or unspecified behavior. Your task is to figure out what happens when you compile and run the program in each puzzle on a conforming C++ implementation.

Take this imaginary puzzle as an example. It's a complete C++ program with a `main` function:

```
#include <iostream>

int main()
{
    std::cout << (1 < 2);
}
```

Your task will be to read through the code and try to guess what the output will be when the program is compiled and run. Always make sure you give it a proper go before turning the page to look at the answer!

Please note a few technicalities:

- For brevity, I declare `main` with no parameters (no `argc` / `argv`) and don't explicitly return a value. These are all optional, and omitting them makes the puzzles slightly easier to read.
- I use `struct` instead of `class` for all the puzzles. There's no semantic difference—`struct`s just have `public` rather than `private` as the default visibility for members, so we don't have to put `public:` everywhere.
- As always in C++, `bool`s are printed as `1` and `0` by default, not `true` and `false`.

The answer to the puzzle is that `1` is less than `2`, so the program prints `1` (representing `true`).

Getting the answers right is only half the fun, though. The other half is understanding *why* it works the way it does. The puzzles are excuses to learn more about how C++ works under the hood. I encourage you to read the explanations thoroughly enough to understand them.

Undefined Behavior

Some puzzles might have undefined behavior. Undefined behavior is the term used when something bad happens during the execution of a program, which the compiler is unable to (or, more specifically, not required to) detect. For instance, we might access an element past the end of an array, or an arithmetic expression with signed integers might overflow. In these cases, the C++ standard imposes no restrictions on the implementation, and anything can happen, including nasal demons. If a puzzle has undefined behavior, your task is to identify the undefined behavior but also to guess what happens in practice on a typical system. Does it actually make demons fly out of your nose, or does something specific happen?

Again, let's look at an example:

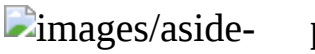
```
#include <iostream>
#include <limits>

int main()
{
std::cout << std::numeric_limits<int>::max() + 1;
}
```

Signed integer overflow in arithmetic is undefined behavior, so if you identified that, you're halfway there!

Making assumptions about what will happen in the case of undefined behavior is a bad idea. So let's do that next! For any puzzles with undefined behavior, the other half of the puzzle is to figure out what would happen if you ran the program on your computer. My computer uses two's complement for signed integers (as do all conforming implementations since C++20), and my CPU doesn't generate an exception when I overflow. So when I add 1 to the largest positive integer, the value simply wraps around to the smallest negative integer. Since my system uses 32-bit ints, the program prints -2147483648. You don't need to know that exact value,

but if you guessed it would print the smallest negative integer, you solved it!

Don’t Do This at Home



Guessing or testing what happens in the case of undefined behavior is an interesting exercise that can teach you more about how C++ works on your platform. It might also enable you to recognize certain error patterns when you see them happening in real programs. But do not make any assumptions about your real programs based on what you find! Your assumptions might be untrue on other computers, after upgrading your compiler, or when you compile with different optimization settings. The compiler is even allowed to remove error checking from your code if it can prove that there’s undefined behavior!

Unspecified and Implementation-Defined Behavior

The C++ standard doesn’t specify everything strictly; it leaves some freedom to the implementation. These are some examples:

- The specific sizes of integer types
- The order of evaluation of function arguments
- The order of initialization of global variables

This allows each implementation to make choices that make the most sense on that particular system.

Most programs have some unspecified or implementation-defined behavior; this is not a bug. And contrary to undefined behavior, demons will not fly out of your nose. It’s just that different implementations might behave a bit differently within a set of allowed behaviors.

If a puzzle has unspecified or implementation-defined behavior, try to also guess what a typical behavior would be.

Playing Around with the Code

The most important part of learning anything programming related is to play around with it yourself. The code from this book can be found on the book's home page at The Pragmatic Bookshelf.[4] You can build it locally by opening `CMakeLists.txt` in your favorite IDE or on the command line:

```
mkdir build
cd build
cmake ..
cmake --build .
```

The project contains one `.cpp` file per puzzle, each resulting in one binary, both named the same as the corresponding puzzle in the book.

You can also try the code directly in your browser by pasting it into an online compiler. I highly recommend Compiler Explorer,[5] where you can choose from and compare different compiler versions and architectures, try out different optimization levels, and add other compiler flags, sanitizer options, and so forth.

Let's get started! Oh, and beware of the demons.

FOOTNOTES

[3] http://catb.org/jargon/html/N/nasal-demons.html

[4] https://pragprog.com/titles/akbrain

[5] https://godbolt.org

Copyright © 2024, The Pragmatic Bookshelf.

Part 1

C++ Brain Teasers

Puzzle 1How Many Degrees?

how-many-degrees.cpp

```
#include <iostream>

struct Degrees
{
    Degrees() : degrees_(0)
    {
        std::cout << "Default constructed\n";
    }
    Degrees(double degrees) : degrees_(degrees)
    {
        std::cout << "Constructed with " << degrees_
<< "\n";
    }
    double degrees_;
};

struct Position
{
    Position() : latitude_{1} { longitude_ =
Degrees{2}; }
    Degrees latitude_;
```

```
    Degrees longitude_;
};

int main()
{
    Position position;
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Constructed with 1
Default constructed
Constructed with 2
```

Discussion

The struct `Position` has two members of type `Degrees`. One is initialized in the member initializer list, and one in the constructor body. So why do we see *three* `Degrees` objects being constructed?

All class members are initialized before we get to the constructor body. If you explicitly initialize the member, as we do for `latitude_`, that's the initialization that'll be used. If you don't, as is the case for `longitude_`, the member will instead be default-initialized, which is why we see the line `Default constructed` being printed. After all members have been initialized, we finally enter the constructor body, initialize a temporary `Degrees{2}`, and then copy-assign it to the previously default-initialized `longitude_`.

Note that instead of explicitly initializing the member in the member initializer list, you can use a default member initializer directly in the member declaration, like so:

examples/default-member-initializer/default-member-initializer.cpp

```
struct Position
{
    Position() { longitude = Degrees{2}; }
    Degrees latitude{1}; // Default member
initializer
    Degrees longitude;
};
```

Using a default member initializer can be a good idea if you have several constructors. Then you don’t have to remember to include that member in every member initializer list.

Further Reading

Constructors and Member Initializer Lists
https://en.cppreference.com/w/cpp/language/constructor

Puzzle 2String Theory

string-theory.cpp

```
#include <iostream>
#include <string>

void serialize(const void*) { std::cout << "const
void*"; }

void serialize(const std::string&) { std::cout <<
"const string&"; }

int main()
{
    serialize("hello world");
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
const void*
```

Discussion

Why does passing a string to `serialize` cause the overload taking a void pointer to be called rather than the overload taking a string?

When we're calling a function with multiple overloads, the compiler uses a process called *overload resolution* to figure out which one is the best fit. The compiler does this by attempting to convert each function argument to the corresponding parameter type for each overload. Some conversions are better than others, and the best conversion is if the argument is already of the correct type.

All the overloads where all arguments can be successfully converted are added to a set of *viable functions*. Then the compiler needs to figure out which overload to select from this set. If an overload has a better conversion for at least one argument and not a worse one for any of the other arguments, this overload is deemed to be the *best viable function* and is selected by overload resolution. If no overload is better than all the others, the call is ill-formed and fails to compile.

Have a look at this example:

```
serialize(int, int); // 1

serialize(float, int); // 2
```

Given these two overloads, suppose you call serialize like this:

```
serialize(1, 2);
```

Both overloads of `serialize` are viable. But the first overload has a better conversion for the first argument (`int` → `int` is better than `int` → `float`) and not a worse conversion for the second argument (`int` → `int` for both overloads), so it is selected by overload resolution as the best viable function.

The puzzle is a bit simpler than this example since both overloads of `serialize` only have one parameter. The first takes a `const void *` and the second takes a `const std::string&`. What does the conversion look like for each of the overloads?

`std::string` is a class in the standard library. It'll typically allocate memory on the heap (unless the string is very small) and allows the string to grow or be otherwise modified at runtime.

However, the string `"hello world"` is not a `std::string` but a simple string literal. String literals are plain C-style arrays of `char`s which get baked into your binary

by the linker and cannot be modified at runtime. A string literal has the type "array of `n const char`." `"hello world"` has 11 characters plus a terminating `\0`, so its type is "array of `12 const char`."

Since the argument `"hello world"` is neither a `const void*` nor a `std::string` but an "array of `12 const char`," a conversion is needed for both overloads. If an implicit conversion exists from the argument to the parameter type, that overload is added to the set of viable functions. Otherwise, the overload is ignored.

Let's examine the first overload and see if "array of `12 const char`" can be implicitly converted to `const void *`. The first thing that happens is that the array gets converted to a pointer. Any "array of `N T`" can be converted to a "pointer to `T`" pointing to the first element. So now our "array of `12 const char`" has turned into a "pointer to `const char`.''

Next, any "pointer to *cv* `T`" (where *cv* means `const`, `volatile`, `const volatile`, or none of these) can be converted to "pointer to *cv* `void`." So now our "pointer to `const char`" has turned into a "pointer to `const void`," which is exactly what the first overload expects.

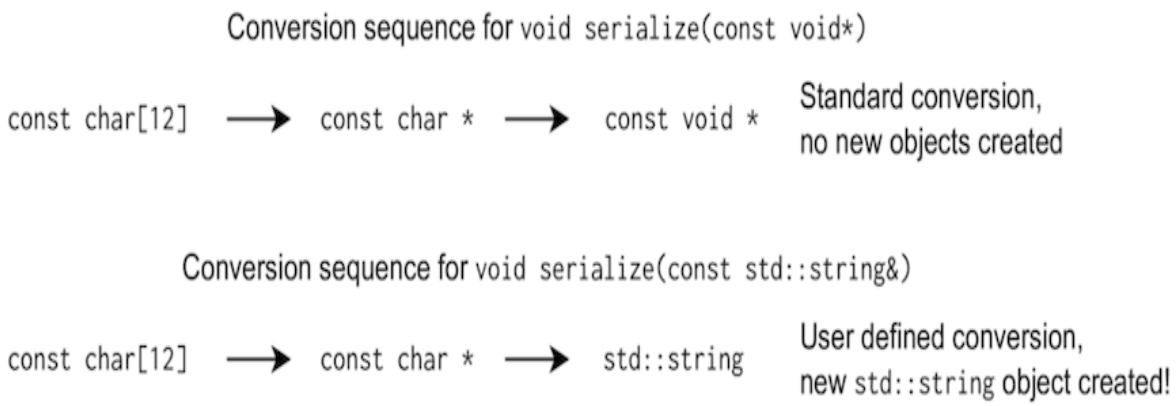
Notice that no constructors or conversion functions were involved in this conversion sequence. This means it's a *standard conversion sequence* and not a *user-defined conversion sequence*. That gets important later.

Let's now examine the second overload, and see if our "array of `12 const char`" can be converted to a "reference to `const std::string`." `std::string` has a constructor `std::string(const char* s)`, which we can use. First, we convert the "array of `12 const`

`char`" to a "pointer to `const char`" as we did above. Then we pass this to the `std::string` constructor and get a `std::string` back, containing a copy of the string literal. The `const std::string&` parameter can bind directly to our `std::string` argument.

Notice that we had to use a constructor for this. This means it's a user-defined conversion sequence and not a standard conversion sequence. It doesn't matter that `std::string` is a standard library type; it still counts as user-defined. The rules are the same for you and the standard library.

Now the compiler has found a valid conversion sequence from our "array of `12 const char`" to the parameter type of each overload and has to figure out which sequence is best:



We only require a standard conversion sequence (top), to call the `const void*` overload. To call the `std::string` overload, we need a user defined conversion sequence (bottom), which involves creating a new temporary `std::string` object. A standard conversion sequence is always *better* than a user-defined conversion sequence, so the first overload gets called, and `const void*` is printed.

Further Reading

Overload Resolution

https://en.cppreference.com/w/cpp/language/overload_resolution

std::string

https://en.cppreference.com/w/cpp/string/basic_string

String Literal

https://en.cppreference.com/w/cpp/language/string_literal

Puzzle 3Hack the Planet!

hack-the-planet.cpp

```cpp
#include <iostream>

int getUserId() { return 1337; }

void restrictedTask1()
{
    int id = getUserId();
    if (id == 1337) { std::cout << "did task 1\n"; }
}

void restrictedTask2()
{
    int id;
    if (id == 1337) { std::cout << "did task 2\n"; }
}

int main() {
    restrictedTask1();
    restrictedTask2();
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program has undefined behavior! But it might display the following output:

```
did task 1
did task 2
```

Discussion

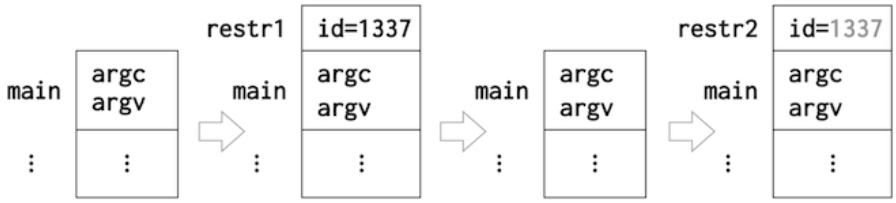
The variable `id` in function `restrictedTask2` has not been initialized and has an indeterminate value. To use its value is undefined behavior. Anything can happen when a program runs into undefined behavior; the C++ standard makes no guarantees. Even the part of the program that happened *before* we took `id`'s value is undefined!

However, if you run this program on your own computer, it will probably print both `did task 1` and `did task 2`, at least if you compile without optimizations. So the

value `1337` magically teleported from `restrictedTask1` to `restrictedTask2`! How could this happen?

Most systems use a stack for local variables. `restrictedTask1` has a local variable `id`, which it sets aside space for in its stack frame. As it happens, `restrictedTask2` has the same number and types of local variables (one `int`), so its stack frame layout will be identical to `restrictedTask1`'s.

When in `main`, the stack will have grown to a certain point—see the leftmost illustration in the following figure. We then call `restrictedTask1`, the stack grows, `restrictedTask1` sets aside space in its stack frame for `id`, and initializes `id` to `1337`. Then control returns to `main` and the stack shrinks again.



Next, we call `restrictedTask2`, the stack grows again, and `restrictedTask2` sets aside space in its stack frame for `id` but doesn't initialize it. However, `restrictedTask2`'s stack frame ends up in the exact same place as `restrictedTask1`'s stack frame. The stack isn't cleared between function calls, so the contents of `restrictedTask2`'s stack frame will be exactly as `restrictedTask1` left it, including the value `1337` in the position of the local variable `id`. This causes `id` to be `1337` in both functions.

If you turn on optimizations, though, `restrictedTask1` and `restrictedTask2` are likely to be inlined into `main`, and the stack is not used for these calls. You can try this yourself by compiling with `-O2` (GCC/Clang) or `/O2` (MSVC). On my x86_64

Linux machine, GCC prints `did task 1` with `-O2`, whereas Clang segfaults. What happens on your machine?

Avoiding Uninitialized Variables

To avoid nasal demons, security holes like this one, and garbage data in general, you should always initialize your variables before using them. It's easy to slip up, but help is to be had! First of all, always compile with warnings. GCC, Clang, and MSVC will warn you about this particular case, as will tools like clang-tidy. But that's only because it's straightforward to prove that `id` is uninitialized when it's used. Often, we can't know this until runtime. This is where sanitizers come in.

Different sanitizers exist for different purposes, but all monitor your code at runtime in various ways to detect problems that can't be detected at compile time. One such sanitizer is MemorySanitizer, which is designed to catch usages of uninitialized memory. If we run this program with MemorySanitizer (pass `-fsanitize=memory` as a compiler option to Clang), it'll print something like this:

```
==1416660==WARNING: MemorySanitizer: use-of-
uninitialized-value
#0 0x5603d43af5eb in restrictedTask2()
example.cpp:14:9
#1 0x5603d43af64d in main example.cpp:19:5
```

It tells you that you're using uninitialized memory and gives you a stack trace to where it happened.

Sanitizers can make your program much slower, so you typically want separate build configurations with sanitizers. A sanitizer can only detect an issue if that issue actually occurs at runtime, though, so make sure to run as much of your test suite as possible with this build.

Recommendations

- Always enable warnings. `-Wall -Wextra -Wpedantic` is a good start on GCC/Clang, and `/W4` on MSVC.
- If your IDE supports integrations with linting tools like clang-tidy, turn them on. They can sometimes report issues like this one before you even compile the code.
- Enable warnings as errors, at least in your Continuous Integration jobs, so you don't overlook any warnings.
- Set up sanitizer builds for as many sanitizers as you can, and run your tests with these builds.

Further Reading

Undefined Behavior

https://en.cppreference.com/w/cpp/language/ub

Deep C (and C++) by Olve Maudal and Jon Jagger

https://www.slideshare.net/olvemaudal/deep-c

List of Sanitizers

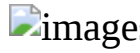
https://github.com/google/sanitizers

Puzzle 4 Going Global

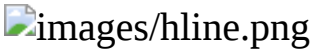
going-global.cpp

```
#include <iostream>

int id;

int main()
{
std::cout << id;
}
```

Guess the Output

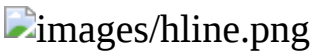


Try to guess what the output is before moving to the next page.



The program displays the following output:

```
0
```



Discussion

This puzzle looks similar to Puzzle 3, Puzzle 3, *Hack the Planet!*; they both use the value of an uninitialized variable id. Unlike Puzzle 3, however, this is not undefined behavior, and the program prints 0. Why is that?

The difference is that the id's in Puzzle 3 are local variables, which pop in and out of existence each time the functions are called. They're said to have automatic storage duration. The id in this puzzle, on the other hand, is a

global variable. More precisely, id is a non-local variable with static storage duration; that is, the lifetime of id is the entire duration of the program.

Unlike variables with automatic storage duration, variables with static storage duration are initialized as a consequence of program initiation. In the simple case of an int with no initializer, it's zero-initialized. If id had an initializer, for example, int id = 2;, we would instead say that it was constant-initialized to 2. (The collective term for zero-initialization and constant-initialization is static initialization.)

Why are variables with static storage duration and no initializer initialized to zero, when variables with automatic storage duration are not? In the general case, initializing a variable isn't free. Some code needs to exist to do said initialization. If, for instance, you define a local variable int id = 0; in a function, most systems need to emit a bit of code to set it to zero every time that function is called. C++ cannot guarantee that this is free or of negligible cost, and a design goal of C++ is to not pay for what you don't use. C++ doesn't want to leave room for a more performant language to exist. So if you decide to omit the initializer and just define int id;, C++ won't make you pay that price.

Another difference between local and non-local variables is their scope. A local variable can only be accessed in the block in which it's defined. A non-local variable, on the other hand, can be accessed from anywhere in the file, or, if declared in a header file, from anywhere in the entire program where that header can be included. So if you omit the initializer for a local variable, you have perfect control over where it will be used next. You probably left it uninitialized because you'll initialize it in the following line or two anyway. But if you omit the initializer for a non-local variable, it's a lot harder to reason about who will access it next.

Compare non-local variables with static storage duration to local variables with automatic storage duration:

	Cost of Initialization	**Scope**

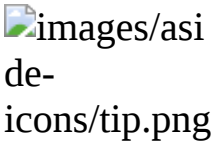
	Cost of Initialization	Scope
Non-local static	Once	The whole file/program
Local automatic	Every time the block is entered	The block in which it's defined

It's harder to control usages of non-local variables with static storage duration, and initializing them is cheap, so C++ will do it automatically for you.

How cheap is it, though? Let's look at the x86 architecture for an example. I said previously that you pay the price of initialization once, but in fact you don't pay it at all! When your program starts, it needs to allocate memory for things like the stack, the heap, the program code, and, importantly for us, global data. On systems like Linux and Windows, when we get fresh physical memory from the operating system, it's filled with zeroes to prevent you from reading the memory of whichever program previously used the same memory page. The compiler/linker can group all uninitialized global variables in one section called .bss. When the program starts, and memory is allocated for the .bss section, it's already filled with zeroes. So these variables are zero-initialized at zero cost!

Note

Note



The word static has many uses in C++. Whether or not id has a static specifier doesn’t affect its static storage duration. Writing static int id; instead of just int id; for a global variable only gives it internal linkage; that is, it cannot be linked to from other translation units (other cpp files). It has static storage duration regardless.

Further Reading

Zero-Initialization

https://en.cppreference.com/w/cpp/language/zero_initialization

More About Initialization of Global Variables: From Program to Process

https://www.youtube.com/watch?v=fGnbGX88z3Y

Puzzle 5A Destructive Relationship

a-destructive-relationship.cpp

```
#include <iostream>
#include <memory>

struct Widget
{
    virtual void draw() { std::cout << "Widget draw\
"; }
    virtual ~Widget() { std::cout << "Widget
destructor\n"; }
};

struct Button : public Widget
{
    void draw() override { std::cout << "Button draw
\n"; }
    ~Button() override { std::cout << "Button
destructor\n"; }
};

int main()
{
```

```
    std::unique_ptr<Widget> widget =
std::make_unique<Button>();
    widget->draw();
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Button draw
Button destructor
Widget destructor
```

Discussion

When we call `draw`, the program only prints `Button draw`. But when the `unique_ptr` goes out of scope and the destructor is called, the program prints *both* `Button`

`destructor` and `Widget destructor`. Why is that?

A common technique for dynamic polymorphism is using virtual functions. You declare a class with one or more functions marked as `virtual`, allowing other classes to override those functions. When you then call a virtual function on a base class pointer pointing to an object of a derived class, the object's dynamic type is used, and the overridden function is called instead.

In this example, imagine a UI framework with a `Widget` class with some common behavior and concrete UI elements like `Button`, `Text`, and `Image`, which override the drawing behavior in the virtual `draw` function.

When we call `draw` on the base class pointer to `Widget`, which is actually pointing to a derived class `Button`, `Button::draw` is called instead of `Widget::draw`, and `Button draw` is printed. However, for the destructor, the behavior is different; we can see that it first prints `Button destructor` and then *also* prints `Widget destructor`. What's going on?

When an object is destroyed, all parts of it have to be destroyed. In this case, `Button` inherits from `Widget`, so each `Button` object consists of a `Widget` object in addition to any members from `Button` (none, in this case). We say that `Button` has a `Widget` *subobject*. If `Button` had a member of type `Widget` instead of inheriting from it, it would also have a `Widget` subobject. In either case, the subobjects must also be destroyed when the complete object is destroyed to avoid resource leaks. Luckily, C++ handles this for us; we don't need to manually call the `Widget` destructor from the `Button` destructor.

Note also that the destructor's body executes first, and only then are the subobject destructors called. This ensures that you can still use members and base classes in your class's destructor. So the program first prints `Button destructor` and then `Widget destructor`.

Bonus Question

What would happen if `Widget` did not declare its destructor as `virtual`, and in which scenarios would that be a problem?

According to the C++ standard, if the static type of the object to be deleted is different from its dynamic type, the static type must be a base class of the dynamic type of the object to be deleted, and the static type must have a virtual destructor. Otherwise, the behavior is undefined.

So if `Widget` doesn't declare its destructor virtual, and you delete a `Button` object through a pointer to `Widget,` the behavior is undefined. In practice, you'll typically see the `Widget` destructor being called instead of the `Button` destructor and have issues like these:

- Any members of the derived class are not destroyed, potentially causing memory leaks or other resource leaks.

- If the derived class has a user-defined destructor, the correctness of the program typically relies on this logic being executed when the object is destroyed.

But in any case, even if none of those are a problem for your program in particular and everything seems fine during testing, you should still declare the destructor virtual to avoid undefined behavior.

Recommendation

Recommendation
Thanks to the automatic destruction of subobjects, a good technique to handle resources in C++ is to have Resource Acquisition Is Initialization (RAII) types (for instance, `unique_ptr`s) as members. This technique also works if you derive from such a class.

Further Reading

CppReference on Destructors

https://en.cppreference.com/w/cpp/language/destructor

The C++ Standard on Destructors and Subobjects

https://timsong-cpp.github.io/cppwp/std20/class.dtor#14

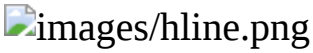
Puzzle 6 Who’s Up First?

whos-up-first.cpp

```
#include <iostream>

struct Resource
{
Resource()
{
std::cout << "Resource constructed\n";
}
};

struct Consumer
{
Consumer(const Resource &resource)
{
std::cout << "Consumer constructed\n";
}
};

struct Job
{
Job() : resource_{}, consumer_{resource_} {}
Consumer consumer_;
Resource resource_;
};

int main() {
Job job;
}
```

Guess the Output

Try to guess what the output is before moving to the next page.



The program displays the following output:

```
Consumer constructed
Resource constructed
```

Discussion

The class Job has two members, consumer_ and resource_, declared in that order. However, in the constructor's member initializer list (the : resource_{}, consumer_{resource_} part), they are initialized in the opposite order. In which order are they actually initialized?

C++ always initializes members in the order they are declared in the class, not in the order they are listed in the constructor's member initializer list. This ensures that members are always initialized in a deterministic order, even if different constructors happen to list them in different orders in their member initializer lists. So even if resource seems to be initialized first in the constructor, consumer_ is actually initialized first.

Having members initialized in a deterministic order is essential if one member depends on the other. For instance, in this example the consumer depends on the resource in its constructor. Everything seems to be okay in the constructor, but since consumer_ is actually initialized before resource_, the Consumer constructor gets passed a Resource that hasn't been constructed yet. If we tried to use it for anything, we would have undefined behavior.

The reverse is also true. Members are destroyed in the opposite order of their declarations in the class. Destruction order can be as important as construction order since one member can depend on another in its destructor, or there can be other lifetime dependencies. Imagine, for instance, the Consumer in this puzzle runs a thread that uses Resource, and that Consumer joins the thread in its destructor. In the current version of the code, resource_ would be destroyed while the thread is still running, and we would risk using a destroyed resource_, resulting in undefined behavior. If consumer_ had been declared after resource_, however, the thread in consumer_ would have stopped before resource_ was destroyed.

Listing the members in the wrong order in the member initializer list can be confusing to the reader, so I recommend always using the same order in the member initializer list and the class declaration. GCC, Clang, and MSVC all have warnings (-Wreorder, -Wreorder-ctor, and C5038, respectively) that warn you if you don't use a consistent order. These warnings are enabled if you compile with -Wall / /Wall, which I always recommend. Different compilers often have different warnings, so compiling with several can be a good idea. That can also help you avoid accidentally using non-standard, platform-specific variations of C++, which would make it harder to port your project to other platforms later.

Recommendations

- Always enable warnings.
- Try compiling with several compilers.

Further Reading

Constructors and Member Initializer Lists

https://en.cppreference.com/w/cpp/language/constructor

Puzzle 7 All Good Things Must Come to an End

all-good-things-must-come-to-an-end.cpp

```
#include <iostream>
#include <string>

struct Connection
{
    Connection(const std::string &name) : name_(name
    {
        std::cout << "Created " << name_ << "\n";
    }

    ~Connection() {
        std::cout << "Destroyed " << name_ << "\n";
    }

    std::string name_;
};

Connection global{"global"};

Connection &get()
{
```

```
    static Connection localStatic{"local static"};
    return localStatic;
}

int main()
{
    Connection local{"local"};
    Connection &tmp1 = get();
    Connection &tmp2 = get();
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Created global
Created local
Created local static
```

```
Destroyed local
Destroyed local static
Destroyed global
```

Discussion

In Puzzle 3, Puzzle 3, *Hack the Planet!*, and Puzzle 4, Puzzle 4, *Going Global*, we learned about the lifetime and initialization differences between a global `id` and a local `id`:

```
int id; // global variable, one instance throughout the whole program

void f() {
    int id = 2; // local variable, created anew each time `f` is called
}
```

In this puzzle, we add two new interesting aspects to object lifetime:

- We no longer initialize the variables to a simple constant; there is a constructor with a side effect.
- We introduce a local variable `localStatic` with static storage duration.

So how does the behavior change when we introduce a constructor and a local static variable? Let's first consider the global variable `global`. Like the global variable `id` we saw previously, it has static storage duration, which means only one instance is alive from its initialization until its destruction at the very end of the program.

We previously saw that the global `int id;` without an initializer was zero-initialized. We also mentioned that a global with an initializer `int id = 2;` would be constant-initialized. Both are examples of static initialization, which happens as part of program initiation. The global variable `global`, on the other hand, is initialized with a non-constant expression. The `Connection` constructor is not `constexpr` (it can't be since it prints to `std::cout`). Objects that can't be statically initialized are instead dynamically initialized, which happens later.

When does dynamic initialization happen, though? It's commonly believed that dynamic initialization happens before `main`—and for good reason: this is a simple and common way to implement dynamic initialization. For instance, Linux has a function `__libc_start_main`, which gets called before `main`. `__libc_start_main` invokes global constructors before eventually calling your `main` function.

The C++ standard doesn't require dynamic initialization to happen before `main`, it merely allows it. However, the standard does require dynamic initialization of global variables to happen before any functions defined in the same translation unit (TU) are called. (A translation unit is just a single `.cpp` file after all the

`#include` files have been included.) So in this puzzle, where `main` is defined in the same TU as `global`, `global` *is* actually guaranteed to be initialized before `main`!

The complete rules for initialization order are rather long and complicated. CppReference.com has an introduction,[6] and the C++ standard has all the details in [basic.start].[7]

Next, let's look at `localStatic`. Like `global`, `localStatic` also has static storage duration, so there's only one instance of `localStatic`, which is alive from its initialization until the end of the program. Unlike globals, local static variables are not initialized when the program starts but rather the first time control passes through their declarations. So even though `localStatic` has static storage duration, it's not initialized until the first time `get` is called.

Finally, `local` is a simple local variable with automatic storage duration; it is initialized each time control passes through its declaration.

Let's look at the execution of this program then:

- The program starts, and right before `main` is called, `global` is initialized.
- We enter `main`, and `local` is initialized.
- We call `get` for the first time, and `localStatic` is initialized.
- We call `get` a second time, but since `localStatic` is already initialized, it's not initialized again.

But we're not done yet! We also have to figure out when our three objects are destroyed. Luckily, the rules for destruction are a lot simpler. Local objects with

automatic storage duration are destroyed at the end of the scope in which they were declared. Objects with static storage duration are destroyed at the end of the program in the reverse order of their creation:

- When we exit `main`, `local` goes out of scope and is destroyed.
- `localStatic` was the last object with static storage duration to be initialized, so it's destroyed first.
- `global` was initialized first and is destroyed last.

Using a static local instead of a global is a good way to get better control over the lifetime of a "global" variable. Instead of referring to a global object each time you need it, you call a function to get a reference to it, and now you have control over when the object is initialized! This can be particularly useful if you have multiple globals depending on each other.

Finally, note a word of caution regarding globals defined in multiple TUs. Although there's some order to the initialization of globals inside a *single* TU, globals in *different* TUs can be initialized in any order. For instance, you might expect the following program to print `1`, but on my computer it prints `0`.

examples/siof/Value.h

```
struct Value
{
    Value(int i_) : i(i_) {}
    Value(const Value &other) : i(other.i) {}
    int i;
```

```
};
```

examples/siof/source1.cpp

```
#include "Value.h"

#include <iostream>

extern Value value1;
Value value2{value1}; // No guarantee that `value1` has been initialized yet!

int main()
{
    std::cout << value2.i;
}
```

examples/siof/source2.cpp

```
#include "Value.h"

Value value1{1};
```

This problem of unknown initialization order between globals in different translation units is often referred to as the “Static Initialization Order Fiasco” (SIOF).[8]

Recommendations

- Avoid globals depending on each other, especially if they're defined in different files.
- Use clang-tidy; it has a check `cppcoreguidelines-interfaces-global-init`, which can detect some uses of dependent globals.
- Instead of global variables, use functions with local static variables to control lifetimes.

Further Reading

More About Dynamic Initialization: How to Start a Program
https://www.youtube.com/watch?v=OGPmZzhDPYw

CppReference.com on Initialization
https://en.cppreference.com/w/cpp/language/initialization

The C++ Standard on Initialization and Termination
https://timsong-cpp.github.io/cppwp/std20/basic.start

Static Initialization Order Fiasco (SIOF)
https://en.cppreference.com/w/cpp/language/siof

Puzzle 8 Will It Move?

will-it-move.cpp

```
#include <iostream>

struct Member
{
};

struct WillItMove
{
    WillItMove() = default;
    WillItMove(WillItMove &&) = default;
    const Member constMember_{};
};

int main()
{
    WillItMove objectWithConstMember;
    WillItMove
moved{std::move(objectWithConstMember)};
    std::cout << "It moved!\n";
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
It moved!
```

Discussion

`WillItMove` has a const member. You can't move from a const object, so why could we still move from `objectWithConstMember`?

In this puzzle, we first initialize a `WillItMove objectWithConstMember;`. Then, on the following line, we initialize a new `WillItMove moved{std::move(objectWithConstMember)};`. `WillItMove` has no copy constructor, so the only alternative to initialize `moved` is the move constructor. But how can the move constructor move from the `const Member constMember`?

Let's start digging from the outside in.

First, what does `std::move(objectWithConstMember)` do? `std::move` doesn't actually move anything; it just turns the lvalue `objectWithConstMember` into an rvalue. Then, this rvalue is used to initialize `moved`.

Lvalues and Rvalues

Lvalues vs. rvalues is a way to categorize expressions. An lvalue is an expression that evaluates to an actual object that has an identity and cannot be moved from. For instance, the expression `objectWithConstMember` evaluates to the `objectWithConstMember` object. It has a name and a place in memory and others could still be using it, so the type system doesn't allow us to move from it. Traditionally, lvalues have tended to appear to the left of an assignment, hence the name lvalue (left-value).

An rvalue on the other hand is an expression that evaluates to something that can be moved from, for instance a temporary that no one else can get to anyway. Rvalues tend to appear to the right of an assignment, hence the name rvalue (right-value).

`std::move` turns an lvalue into an rvalue and is the way we tell the type system that "this is fine, you can go ahead and move from this lvalue; I won't be using it anymore." With the introduction of `std::move`, it has become more common to see lvalues appear to the *right* of the assignment (inside a

Lvalues and Rvalues

> std::move), so the “left-value” vs. “right-value” mnemonic isn’t quite as useful as it used to be.
> See Lvalues, Rvalues, Glvalues, Prvalues, Xvalues, Help![9] for a full overview of lvalues, rvalues, and the other value categories.

Note that even though we initialize `moved` from an rvalue, that doesn’t mean we need to use the move constructor to initialize it! We just need any constructor capable of binding to an rvalue. A const lvalue reference can bind to an rvalue, so a copy constructor could also have been used, if we had one. We don’t, however, so `WillItMove`’s move constructor is our only choice.

`WillItMove`’s move constructor has been explicitly defaulted by the line `WillItMove(WillItMove &&) = default;`, so let’s turn our attention to what a defaulted move constructor does. How can it move from a const member?

The explicitly defaulted move constructor direct-initializes all subobjects with the corresponding member from the source object. Essentially, it’s defined something like this:

examples/default-move/default-move.cpp

```
Class(Class &&other) :
```

```
    BaseClass1{std::move(other)},
    BaseClass2{std::move(other)},
    /* ... */
    member1_{std::move(other).member1_},
    member2_{std::move(other).member2_}
    /* ... */
    {}
```

So for our class `WillItMove`, it looks something like this:

examples/default-move/default-move.cpp

```
WillItMove(WillItMove && other) :
     constMember_{std::move(other).constMember_}{}
```

Again, even if `std::move(other).constMember_` is an rvalue, we don't necessarily need to use a `Member` move constructor to initialize `constMember_`. Either a copy or a move constructor will do, as long as it can bind to `std::move(other).constMember_`. Since we haven't declared any special member functions for `Member`, it implicitly gets both a defaulted copy constructor and a defaulted move constructor. Can one of those bind to the const rvalue `other.constMember_`?

The implicitly defaulted move constructor for `Member` is declared like this:

```
Member(Member&&)
```

An rvalue reference to non-const cannot bind to the const rvalue, so we can't use the move constructor to direct-initialize `constMember_`.

The implicitly defaulted copy constructor for `Member` is declared like this:

```
Member(const Member&)
```

An lvalue reference to const can bind to a const rvalue, so we can use the copy constructor to direct-initialize `constMember_`.

Thus, the explicitly defaulted move constructor for `WillItMove` uses `Member`s *copy* constructor rather than `Member`s move constructor to initialize `constMember_`.

Further Reading

Move Constructors
https://en.cppreference.com/w/cpp/language/move_constructor

Copy Constructors
https://en.cppreference.com/w/cpp/language/copy_constructor

Direct-Initialization
https://en.cppreference.com/w/cpp/language/direct_initialization

std::move
https://en.cppreference.com/w/cpp/utility/move

The Difference Between No Move Constructor and a Deleted Move Constructor

https://blog.knatten.org/2021/10/15/the-difference-between-no-move-constructor-and-a-deleted-move-constructor/

Puzzle 9 Counting Copies

counting-copies.cpp

```
#include <iostream>

struct Resource
{
    Resource() = default;
    Resource(const Resource &other)
    {
        std::cout << "copy\n";
    }
};

Resource getResource()
{
    return Resource{};
}

int main()
{
    Resource resource1 = getResource();
    Resource resource2{resource1};
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
copy
```

Discussion

It may seem like three copies are being made in this program. One to initialize the return object of `getResource` with the temporary `Resource{}`, one to initialize `resource1` with said return object, and one to initialize `resource2`. However, no copies are being made when returning from `getResource()` and initializing `resource1`. How is this possible?

Let's first look at the return statement, which seemingly initializes the return object from the temporary `Resource{}`. The expression `Resource{}` in the return statement is a type of rvalue called a *prvalue*, or "pure" rvalue. In the previous puzzle Puzzle 8, *Will It Move?*, we first met lvalues and rvalues and learned that

rvalues are expressions you can move from, bind rvalue references to, and so on. In fact, there are two types of rvalues—xvalues and prvalues:

- An xvalue (or "eXpiring lvalue") is what you get when you call `std::move` on an lvalue. For instance, in Puzzle 8, *Will It Move?*, the expression `objectWithConstMember` is an lvalue. It evaluates to an actual object that exists somewhere, that we can take the address of, and so on. When we do `std::move(objectWithConstMember)`, we now have an xvalue. This expression still evaluates to an actual object, but we're also allowed to treat it as an rvalue and move from it.
- A prvalue, on the other hand, is a "pure" rvalue. It's an expression that initializes a new object, as opposed to just turning an existing object into an rvalue.

In C++11, prvalues were unnamed temporary objects, and the compiler was allowed to elide, or omit, copies of them. The form of copy elision where the unnamed temporary was the operand of a return statement, like in this puzzle, was called *unnamed return value optimization* (URVO)—unnamed because it applied when the function returned an unnamed temporary. In C++17, avoiding copies in these situations became mandatory. But rather than making URVO mandatory, the language was changed in such a way that now there's never even an object here to optimize away the copy of in the first place! Prvalues are no longer unnamed temporaries; they merely represent the *idea of an object*, which will only materialize later.

Another form of return value optimization, which is still not mandatory, is called *named return value optimization* (NRVO). This applies when you return a local variable rather than a prvalue; in this case, the implementation is *allowed* to omit the copy but isn't required to. NRVO applies even if the copy constructor has side effects; these side effects then simply don't happen. This is a very rare case where an optimization can change your program's observable behavior!

Here's an example of both unnamed and named return value optimization:

examples/rvo/rvo.cpp

```
Resource getResource() {
    return Resource{}; // Unnamed RVO (pre C++17)
}
```

examples/rvo/rvo.cpp

```
Resource getResource()
{
    Resource resource;
    return resource; // Named RVO
}
```

Let's now look at the expression `Resource resource1 = getResource()`, which seemingly copies the returned object into `resource1`. A function call is a prvalue, so the same idea applies here. The call to `getResource()` does not result in a

temporary object; it merely represents the *idea* of the return value, and no `Resource` object is actually created until `resource1` is initialized with it.

Defining things like "the idea of an object" in the standard is all well and good, but how is this possible to do in practice? After all, the `Resource` object is supposed to end up on the stack in `main`, but the actual code that creates it is in `getResource()`, which does not have access to `main`'s stack frame.

The trick that most systems use to avoid the copy is that `main` sets aside space in its stack frame for `resource1`, just like for any other local variables. Then `main` passes a pointer to this memory as a "secret" extra argument to `getResource()`, as if its signature was `getResource(Resource* ptr)`. Now `getResource()` can construct the object directly in this memory location.

For this to work, the caller and the callee must agree on this convention, so this is part of the system's Application Binary Interface (ABI). One popular ABI is the Itanium C++ ABI,[10] which, for instance, is used on Linux. GCC and Clang both conform to this ABI, so you can freely call a function compiled with one from a function compiled by the other. The Itanium C++ ABI says:

> *If the return type is a class type that's non-trivial for the purposes of calls, the caller passes an address as an implicit parameter. The callee then constructs the return value into this address.*

If we try this out on Compiler Explorer, choosing the x86_64 architecture, we find the current versions of GCC and Clang both generate this code in `main`:

```
lea     rax, [rbp-4]
mov     rdi, rax
call    getResource()
```

`[rbp-4]` is the address of `resource1` in `main`'s stack frame. This address gets loaded in `rdi`, the register used to pass the first integer/pointer argument to a function, and then `getResource()` is called.

For MSVC, it's harder to track down information about their ABI. But using Compiler Explorer, we can see a similar pattern on x86_64:

```
lea     rcx, QWORD PTR resource1$[rsp]
call    Resource getResource(void)   ; getResource
```

The convention is the same except the register used for the pointer is `rcx`.

Recommendation
All the mainstream compilers have done both named and unnamed

Recommendation

return value optimization for many years now (I checked[11] as far back as Clang 3, GCC 4, and Visual Studio 2008), so the takeaway is to simply return by value and not try to do something clever to “optimize” your returns.

Further Reading

Blog Post from 2011 Demonstrating How Mainstream Compilers Were Already Then Doing Copy Elision Very Well
https://blog.knatten.org/2011/08/26/dont-be-afraid-of-returning-by-value-know-the-return-value-optimization/

Guaranteed Copy Elision Does Not Elide Copies
https://blog.tartanllama.xyz/guaranteed-copy-elision/

Return Value Optimization/Copy Elision
https://en.cppreference.com/w/cpp/language/copy_elision

The Itanium C++ ABI
https://itanium-cxx-abi.github.io/cxx-abi/abi.html

All the Assembly in This Example Explained
https://www.youtube.com/watch?v=I7j6QCO8xMc

Puzzle 10 A Strange Assignment

a-strange-assignment.cpp

```cpp
#include <iostream>
#include <string>

std::string getName()
{
    return "Alice";
}

int main()
{
    std::string name{"Bob"};
    getName() = name;
    std::cout << "Assigned to a function!\n";
}
```

Guess the Output

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Assigned to a function!
```

Discussion

Wait, what? You can assign to a function? What does that even mean?

At first glance, it looks like we're assigning a variable to a function call, which is the opposite of what we typically see (`name = getName()`). Let's have a closer look at what's going on.

The expression `getName()` calls `getName`, which returns a string `"Alice"`. This results in a temporary object of type `std::string` with the value `"Alice"`. Then we copy-assign the variable `name` (which contains the value `"Bob"`) into this temporary. When we look at it this way, it suddenly makes sense. But a closer look raises some questions.

In Puzzle 9, Puzzle 9, *Counting Copies*, you were introduced to prvalues:

```
Resource getResource() { return Resource{}; }

Resource resource1 = getResource();
```

There we discussed that the function call `getResource()` is a prvalue, which does not result in a temporary `Resource`. Instead, the return value is created directly in the `resource1` object, and no copy is allowed to be made.

The expression `getName()` in this puzzle is also a prvalue, so the returned object doesn't even exist until we assign it to something, which we don't do!

If no object exists, how can we then copy-assign `name` to it? Temporary materialization to the rescue. The reason that function calls are prvalues and don't create objects at all is to avoid unnecessary copies. In the preceding `Resource` example, creation of an actual object is delayed until we assign the result of `getResource()` to `resource1`. But sometimes we call a function just for its side effects and don't assign the result to anything, like when we call `getName()` in this puzzle. When we don't assign the result of an expression to anything, we call it a discarded-value expression. We still want an object to be created, so a temporary is materialized. That's what happens here—the prvalue expression `getName()` creates a temporary `std::string`.

Now that we have an actual `std::string` temporary, we can copy-assign `name` to it just fine.

Binding References to Prvalues

Another case where temporary materialization is important is when you bind a reference to a prvalue. Imagine, for instance, we had a function like one of these:

```
void useName(const std::string&
name_ref);
void useName(std::string&& name_ref);
```

If we then did `useName(getName());`, the reference parameters `name_ref` would need to have actual objects to bind to. And that's indeed what happens when you bind a reference to prvalue—a temporary is materialized.

Further Reading

Lvalues, Rvalues, Glvalues, Prvalues, Xvalues, Help!
https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/

Puzzle 11What's the Time of Death?

whats-the-time-of-death.cpp

```
#include <iostream>

struct MemoryArea
{
    MemoryArea(int number) : number_(number) {}
    ~MemoryArea() { std::cout << "Freed memory area
<< number_ << "\n"; }

    int number_;
};

MemoryArea getMemory(int number) { return
MemoryArea{number}; }

struct DataSource
{
    DataSource(const MemoryArea &memoryArea)
        : memoryArea_(memoryArea) {}
    const MemoryArea &memoryArea_;
};
```

```
int main()
{
    const auto &reference1 = getMemory(1);
    std::cout << "Bound reference 1\n";
    const auto &reference2 = getMemory(2).number_;
    std::cout << "Bound reference 2\n";
    const auto &reference3 =
DataSource(getMemory(3));
    std::cout << "Bound reference 3\n";
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Bound reference 1
Bound reference 2
Freed memory area 3
```

```
Bound reference 3
Freed memory area 2
Freed memory area 1
```

Discussion

As we've seen a few times by now, a function call is a prvalue, and no object exists until that prvalue is used to initialize one. In the previous puzzle we also saw an exception to this rule: when we didn't use the prvalue to initialize an object, we called that a discarded-value expression and learned that a temporary is then materialized.

In this puzzle, the results of the function call expressions aren't assigned to objects as in Puzzle 9, *Counting Copies*, but neither are they discarded as in Puzzle 10, *A Strange Assignment*. Instead, we're binding references to them in various ways. What happens then?

- For the first call, we bind the reference `reference1` directly to the prvalue. A temporary is materialized, as we briefly touched upon in Puzzle 10.
- For the second call, we access the member `number_` on a class prvalue. This is another case where a temporary needs to be materialized. Otherwise, no object would be there for us to access the member of.
- For the third call, we pass the prvalue to the `DataSource` constructor. This is actually the same case as for the first call; the constructor parameter reference `memoryArea` binds to the prvalue, and a temporary is materialized.

So a temporary is materialized for each of the three calls to `getMemory`. But for how long do they stay alive?

Normally, temporaries get destroyed at the end of the full-expressions in which they were created. In the cases in this puzzle, that means they would be destroyed at the end of the line. However, the lifetime of temporaries is extended in certain cases. This is somewhat of a relic of the past, which is no longer very useful. For instance, for `const auto &reference1 = getMemory(1);`, we'd be much better off just writing `const auto object1 = getMemory(1);`. Thanks to the new rules for prvalues in C++17, there'd still be no extra copy, and we wouldn't be asking all these questions. Fewer questions means better code!

So what exactly are the rules for lifetime extension? Although others exist, by far the most common, and the ones that matter here, are these:

- If you bind a reference to an object you obtained through temporary materialization, that temporary object persists for the lifetime of the reference.
- If you bind a reference to a *member of* an object you obtained through temporary materialization, the full temporary object (not just the member) persists for the lifetime of the reference.

Let's examine each line:

```
const auto &reference1 = getMemory(1);
```

We bind `reference1` to the temporary object we got from `getMemory(1)`. Temporary 1 persists for the lifetime of `reference1`—that is, to the end of `main`.

```
std::cout << "Bound reference 1\n";
```

`Bound reference 1` is printed.

```
const auto &reference2 = getMemory(2).number_;
```

We bind `reference2` to the member `number_` of the temporary object we got from `getMemory(2)`. The full temporary object 2 (not just the member) persists for the lifetime of `reference2`—that is, to the end of `main`.

```
std::cout << "Bound reference 2\n";
```

`Bound reference 2` is printed.

```
const auto &reference3 = DataSource(getMemory(3));
```

We bind the constructor parameter reference `memoryArea` to the temporary object we got from `getMemory(3)`. This is used to create a new object of type `DataSource`. Then we bind `reference3` to the temporary `DataSource` which gets materialized when we call its constructor. The temporary `DataSource` persists for the lifetime of `reference3`—that is, to the end of `main`.

But wait, it's a trap! As soon as the `DataSource` object has been constructed, the `memoryArea` parameter is gone. It doesn't help that we assign it to the `memoryArea_` member; that's a different reference! So the temporary `MemoryArea` 3 does *not* get its lifetime extended, and the memory is freed at the end of the line. `Destroyed 3` is printed, and the `MemoryArea` that the `DataSource` was planning to read from has already been freed and is now a dangling reference.

```
std::cout << "Bound reference 3\n";
```

`Bound reference 3` is printed.

```
}
```

Ahh, `}`, the best feature of C++! At the end of the scope, local variables are destroyed in the reverse order of their construction. First, `reference3` is destroyed, which means the temporary `DataSource` is destroyed (its destructor doesn't print anything). Next, `reference2` is destroyed, which means temporary 2 is destroyed, and `Destroyed 2` is printed. Finally, `reference1` is destroyed, which means temporary 1 is destroyed, and `Destroyed 1` is printed.

Related Issues

Similar issues as for temporary 3 are not uncommon in C++. Here's a famous example that didn't get fixed until C++23. (At

ated Issues

the time of writing, common compilers have yet to implement this fix.)

examples/range-for/range-for.cpp

```
class MemoryAreaContainer
{
  public:
    MemoryAreaContainer();
    std::vector<MemoryArea>
&getMemoryAreas()
    {
        return memoryAreas_;
    }

  private:
    std::vector<MemoryArea> memoryAreas_;
};
```

examples/range-for/range-for.cpp

```
for (const auto &lp :
MemoryAreaContainer{}.getMemoryAreas())
{
    std::cout << lp.number_ << std::endl;

```

Related Issues

```
}
```

This looks harmless. But for the temporary `MemoryAreaContainer` to persist to the end of the for loop, we need to bind a const reference to it. Our const reference here isn't bound to the `MemoryAreaContainer` but rather to the return value from `getMemoryAreas()`. So the `MemoryAreaContainer` temporary gets destroyed before we even start iterating. The reference we got from `getMemoryAreas()` refers to the already destroyed `memoryAreas_` member and is dangling. (This is a bit of a simplification; please refer to stmt.ranged[12] to see the full code that the range-based for statement expands to.)

Recommendations

Recommendations

- Don’t rely on lifetime extensions, prefer `const auto obj = get()` over `const auto& ref = get()`.
- Don’t rely on a lifetime extension surviving through function calls.
- Keep up with new C++ standards, compilers, and tools. Often, bugs get fixed in the compilers or even the language itself.
- Use sanitizers. Address sanitizer in particular can catch some uses of objects that have been destroyed.

Further Reading

Lifetime of a Temporary
https://en.cppreference.com/w/cpp/language/reference_initialization#Lifetime_of_a_temporary

Final Fix of Broken Range-Based for Loop
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2644r1.pdf

Puzzle 12 A False Start

a-false-start.cpp

```
#include <iostream>
#include <stdexcept>

struct Engine
{
    ~Engine() { std::cout << "Engine stopped\n"; }
};

struct Machine
{
    Machine() { throw std::runtime_error{"Failed to start machine"}; }
    ~Machine() { std::cout << "Machine stopped\n"; }

    Engine engine_;
};

int main()
{
    try
```

```
    {
        Machine machine;
    }
    catch (...)
    {
    }
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Engine stopped
```

Discussion

In this puzzle, we have a class `Machine`, which contains an `Engine`. But `Machine` throws in its constructor before it's completely done initializing. Does the engine get stopped? Does the `Machine` itself get stopped?

One of the great things about C++ is that constructed objects get destroyed automatically and deterministically (unless you manually call `new`, which these days should be left to library authors). You'll find no need for Python's `with` or C#'s `using`, and rarely a need for something like `finally`.

But only constructed objects get destroyed, so what does it mean to be constructed? And what happens in the case of "half-constructed" objects like the `Machine` instance here, which throws during construction?

An object is only deemed to be constructed when initialization has completed. The initialization of a class object goes in this order:

- Virtual base classes
- Direct base classes
- Non-static data members (like `Machine::engine_`)
- The constructor body

So when we're constructing the `Machine` object, we first initialize the `Engine` member. As soon as this is done, `engine_` is deemed constructed and will eventually have to be destroyed. Then we get to the `Machine` constructor body, which throws an exception. The initialization of `Machine` did not complete, and it's not deemed constructed. Only the `Engine` destructor gets called, not the `Machine` destructor.

This is very useful since you don't have to think about cleaning up half-initialized objects. Any initialized member like `engine_` gets destroyed automatically; the rest don't need to be since they were never constructed in the first place. You do have to be careful, though, if you manually acquire some resource in the constructor. In that case, you can't rely on the destructor to clean it up. For instance, if you call `new` and assign the pointer to a member, you cannot rely on a `delete` in your destructor to clean it up. The same goes for raw calls to function pairs like `open`/`close`.

As always, to make your code more robust, avoid manually calling functions like `delete` and `close`. Instead, use standard library types like `unique_ptr` or `fstream` as members, which automatically clean up the managed resources in their destructors. If you have several such resource members, the ones that got initialized, and only those, will be automatically released even in the case of a half-constructed object like `machine`.

If your class manages a custom resource that doesn't fit in a standard library type, such as the engine in this puzzle that needs to be stopped, factor that responsibility out to a separate small class like `Engine` instead of handling it manually in the containing class. Then you don't need to think about managing it at all; it will always be destroyed if it was constructed.

Recommendations

Recommendations

- Avoid manually managing resources; use standard library types like `unique_ptr` and `fstream` as members.
- For custom resources, factor out classes whose only responsibility is managing resources of that kind.

Further Reading

Class Initialization Order

https://en.cppreference.com/w/cpp/language/constructor

Puzzle 13 A Constant Struggle

a-constant-struggle.cpp

```cpp
#include <iostream>
#include <type_traits>

template <typename T>
void byValue(T t)
{
    std::cout << std::is_const_v<T>; // true if T is const
}

template <typename T>
void byReference(T &t)
{
    std::cout << std::is_const_v<T>; // true if T is const
}

int main()
{
    int nonConstInt = 0;
```

```
    const int constInt = 0;

    byValue(nonConstInt);
    byValue(constInt);

    byReference(nonConstInt);
    byReference(constInt);
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
0001
```

Discussion

The two function templates `byValue` and `byReference` take their parameters by value and by reference, respectively. We then call these two function templates with a non-`const` and a `const int` as arguments. In which cases is `T` deduced as `const`?

Before we start digging into the deduction rules, let's clarify two concepts that can be easily mixed up. The `T` in `template <typename T>` is called the *template parameter*. The goal of template argument deduction is to find `T`. On the other hand, the `T` and `T &` in the function signatures `byValue(T t)` and `byReference(T &t)` are called the *template parameter types*. These are the actual parameter types to the function templates and are what we use for deduction.

Let's refer to these template parameter types as `P` to distinguish them from the template parameters `T`. Here are three examples:

	Template parameter	Template parameter type `P`
`template <typename T>`		
`void byValue(T t)`	`T`	`T`
`template <typename T>`		
`void byReference(T & t)`	`T`	`T &`

	Template parameter	Template parameter type `P`
`template <typename ValueType>`		
`void f(const std::vector<ValueType>& v)`	`ValueType`	`const std::vector<ValueType>&`

As you can see from the examples, in both `byValue` and `byReference`, the template parameter is `T`. In `byValue`, the template parameter type `P` is also just `T`, but in `byReference`, it is `T &`.

With that out of the way, let's dig into the deduction rules. First, let's look at the calls to `byValue`:

```
template <typename T> void byValue(T t);

int nonConstInt = 0;
const int constInt = 0;

byValue(nonConstInt);
byValue(constInt);
```

Remember, the template parameter is `T`, and the template parameter type `P` is also just `T` (as opposed to, for example, `T &`).

When deducing the template parameter type `P`, if it's not a reference type, the deduction rules say to ignore any top-level cv-qualifiers (`const` or `volatile`) of the argument. Since `P` is just `T`, it's not a reference type, and cv-qualifiers are ignored.

Wait, no matter if we pass a non-`const` or a `const` argument, we pretend it's non-`const`! Isn't that a bit shady? It's not. When `P` is not a reference type, it means we pass the argument by value. It doesn't matter whether the original argument was `const`, the `t` parameter inside `byValue` is a copy, and we can't modify the original argument.

Since top-level cv-qualifiers are ignored, we deduce `P` from a non-`const int` in both calls to `byValue` and find `P == int` in both cases. And since `P` == `T`, `T` is also non-`const`. `0` is printed for both calls.

Next, let's look at the calls to `byReference`:

```
template <typename T> void byReference(T &t);

int nonConstInt = 0;
const int constInt = 0;

byReference(nonConstInt);
byReference(constInt);
```

The template parameter is `T`, and the template parameter type `P` is `T &`.

This time, `P` is a reference, and we do *not* ignore the top-level cv-qualifiers of the argument. In this case, that's important! The reference parameter `t` will bind to our original argument, so it's important that we don't try, for instance, to bind a non-`const` reference to a `const` argument.

When passed the non-`const` `nonConstInt`, `P` is deduced as `int &`, which means that `T` is `int`. It's not `const`, so `0` is printed.

When passed the `const` `constInt`, `P` is deduced as `const int &`, which means that `T` is `const int`. It is `const`, so `1` is printed.

The complete rules for template argument deduction can be a bit daunting to read, but at least this part about ignoring top-level cv-qualifiers is intuitive enough that I'm able to remember it.

Auto

Auto

The deduction rules for `auto` are based on the rules for template argument deduction. So now that you know these template argument deduction rules, you also know the corresponding `auto` deduction rules!

```
auto t1 = nonConstInt;  // non-const object
auto t2 = constInt;     // non-const object
auto &t3 = nonConstInt; // reference to non-const object
auto &t4 = constInt;    // reference to const object
```

Further Reading

Template Argument Deduction

https://en.cppreference.com/w/cpp/language/template_argument_deduction

Deducing Template Arguments from a Function Call

https://timsong-cpp.github.io/cppwp/std20/temp.deduct.call

Puzzle 14 Aristotle's Sum of Parts

aristotles-sum-of-parts.cpp

```cpp
#include <iostream>
#include <type_traits>

int main()
{
    char char1 = 1;
    char char2 = 2;

    // True if the type of char1 + char2 is char
    std::cout << std::is_same_v<decltype(char1 + char2), char>;
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
0
```

Discussion

I know, I know. This is supposed to be a fun book. But sometimes we need to talk about sad topics like this.

As you've figured out by now, the type of `char + char` is unfortunately not `char`. So what is it, then?

For many binary operations, such as `+`, `-`, `*`, `/`, `>`, `==`, `^`, and more, the operands must be of the same type. You don't have to ensure that yourself, though; C++ will take care of it. C++ is in fact very eager to implicitly convert between types when you're not looking, and this is one of the more confusing cases. Meet the usual arithmetic conversions.

The usual arithmetic conversions are performed on the operands to these operators to ensure both operands are of the same type. This common type is also the type of the result. For instance, if you add a `float` and an `int`, the `int` is first converted to a `float`, and the result is a `float`. And if you add a `float` and a `double`, the `float` is first converted to a `double`, and the result is a `double`. Makes sense so far.

The confusion starts when both operands are of integer types, especially if they are of the *same* type. For instance, if you add a `char` and an `int`, the `char` is first converted to an `int`, and the result is an `int`. It still makes sense. But what if you add a `char` and a `short`? Rather than converting the `char` to a `short`, both operands are converted to an `int` (usually—see the following). And it gets worse: if you add two `char`s, which are already of the same type, they're still converted to a different type!

Let's see what the usual arithmetic conversions do to our expression `char1 + char2`. Since both operands are of integer types, a process known as integral promotions is performed on both of them. The idea is to convert smaller integer types to `int` or `unsigned int` before performing the arithmetic expression. After converting both operands, we can now add them using regular `int` or `unsigned int` addition and get an `int` or `unsigned int` as a result.

The rule for promoting a smaller integer type is that it can be converted to an `int` if `int` can represent all the values of the smaller type; otherwise, it can be converted to an `unsigned int`. Does a `char` fit in an `int`? Usually it does, but that depends on the implementation. (I told you this is a sad story.)

A complicating factor here is that the sizes of fundamental types in C++ are not fixed. An implementation can decide that `int` is 16 bits or 64 bits, and even that `char` and `long` are of the same size. The only rules are these:

- Each of `signed char`, `short`, `int`, `long`, and `long long` is at least as large as the preceding type in the list.

- For each of the types in that list, an `unsigned` version exists, which has the same size as the `signed` version.

Some minimum sizes are also defined:

Type	Minimum width
`signed char`	8
`short`	16
`int`	16
`long`	32
`long long`	64

On a typical 64-bit Linux, Windows, or macOS system, `int` is 32 bits, whereas `long` is 64 bits on Linux/macOS and 32 bits on Windows. But a conforming implementation could equally well decide that all integer types are 64 bits.

In this puzzle, we're interested in `char` in particular, but `char` is curiously missing from all the lists above. What's going on? `char` is a bit special in that the implementation can choose whether to back it by `signed char` or `unsigned char`. In either case, `char` is a distinct type. So `char`, `signed char`, and `unsigned char` are

actually three different types, and whether or not a plain `char` is signed is implementation-defined!

Remember, we're interested in whether all the values in `char` fit in an `int` to figure out whether our `char`s would be converted to `int` or `unsigned int`. Normally they all fit, so both operands of `char1 + char2` would be converted to `int`, and the result would also be an `int`.

But let's say that the implementation has chosen `char` and `int` to both be 16 bits. Let's also say that the implementation has chosen that `char` is unsigned. The range of `int` is then -2^{16-1} to $2^{16-1}-1$, or -32768 to 32767, inclusive. The range of `char` is 0 to $2^{16}-1$, or 0 to 65535, inclusive. That is, half the `char`s don't fit in an `int`! On these systems, the operands of `char1 + char2` would instead be promoted to `unsigned int` and the result would also be an `unsigned int`.

So the type of `char1 + char2` can either be `int` or `unsigned int`. All we know for sure is that it's not `char`.

Further Reading

The Usual Arithmetic Confusions
https://shafik.github.io/c++/2021/12/30/usual_arithmetic_confusions.html

64-Bit Data Models
https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models

Fundamental Types
https://timsong-cpp.github.io/cppwp/std20/basic.fundamental

Usual Arithmetic Conversions

https://timsong-cpp.github.io/cppwp/std20/expr.arith.conv

Integral Promotions

https://timsong-cpp.github.io/cppwp/std20/conv.prom

Puzzle 15 Back from the Future

back-from-the-future.cpp

```
#include <future>
#include <iostream>

int main()
{
    char counter = 0;

    auto future1 = std::async(std::launch::async, [&
()
    {
      counter++;
    });
    auto future2 = std::async(std::launch::async, [&
()
    {
      return counter;
    });

    future1.wait();
```

```
    // Cast to int to print it as a numerical value rather than a character
    std::cout << (int)future2.get();
}
```

Guess the Output	
images/aside-icons/important.png	Try to guess what the output is before moving to the next page.

The program has undefined behavior! Importantly, the answer is not “either 0 or 1.”

Discussion

When two unsynchronized threads access the same memory location and at least one of these accesses is a modification, we have a data race. This program launches two threads, one that modifies `counter` and one that reads it. Since we have made no effort to synchronize the two threads, the program has a data race.

If you run this program on your machine, you're very likely to see either `0` or `1`, depending on which thread gets to `counter` first. The `counter` is a single byte, so you're not likely to see, for example, any half-written values. Either the `counter++` operation happens entirely before the second thread reads `counter` or it happens entirely after. But that doesn't mean you can rely on this for some sort of "eventually consistent" logic! A data race results in undefined behavior, so anything can happen. Remember what we talked about in the preface—demons could come flying out of your nose. Don't try to reason about or test what happens under undefined behavior. Your program might seem to work, but break on another system when the compiler is upgraded, or every one-millionth time it runs.

For these threads to safely access `counter`, some sort of synchronization is needed. The classical solution is to protect `counter` with a mutex, but this is cumbersome and also introduces locks. Locks can be slow, and where there are locks, there can be deadlocks. A better approach is to use an atomic, like so:

examples/std_atomic/std_atomic.cpp

```
std::atomic<char> counter = 0;

auto future1 = std::async(std::launch::async, [&]()
                          counter++;
                        });
auto future2 = std::async(std::launch::async, [&]()
                          return counter.load();
                        });
```

Using `std::atomic` instead of a mutex makes the code simpler, avoids introducing an extra variable for the mutex, ensures we can't forget to lock the mutex, and probably makes the code lock-free.

Lock-Free Atomics

Lock-Free Atomics

Whether or not `std::atomic<char>` is lock-free is implementation-defined, but it typically is. You can check this on your system with the `is_lock_free` / `is_always_lock_free` members of the `std::atomic` class template:

examples/std_atomic/std_atomic.cpp

```
static_assert
(counter.is_always_lock_free);
assert(counter.is_lock_free());
```

- `is_always_lock_free` is `true` if `std::atomic<T>` for the given `T` is always lock-free. This is a `static constexpr` member, which can be checked at compile time.
- `is_lock_free()` is `true` if a particular *instance* of `std::atomic<T>` is lock-free and can be `true` even if `is_always_lock_free` is not. Some atomics could, for instance, be lock-free only if they're aligned. This can only be checked at runtime.

Both on my M1 Macbook and my x86_64 Linux machine, `std::atomic` of `int`, `long`, and `double` are all always lock-free.

A great tool to detect data races is the free ThreadSanitizer. It's one of many sanitizers that ship with LLVM, and it's enabled by simply passing the `-fsanitize=thread` compiler option and then running your program. It will tell you something like this for this puzzle:

```
WARNING: ThreadSanitizer: data race (pid=2735)
  Read of size 1 at 0x7ffed85a35cf by thread T2:
    (... stack trace ...)

  Previous write of size 1 at 0x7ffed85a35cf by
thread T1:
    (... stack trace ...)

  Location is stack of main thread.

  Location is global '??' at 0x7ffed8584000
([stack]+0x1f5cf)

  Thread T2 (tid=2738, running) created by main
thread at:
    (... stack trace ...)

  Thread T1 (tid=2737, finished) created by main
thread at:
    (... stack trace ...)
```

As you can see, it not only tells you that there was a data race, but also tells you where the variable is, the stack traces to the code that read and wrote it, and the stack traces to where the threads were created. Pretty useful stuff!

Recommendations

- Always synchronize accesses to shared variables if at least one of them is a write.
- `std::atomic` is often a great choice for simple synchronization of values.
- Use sanitizers.

Further Reading

std::async
https://en.cppreference.com/w/cpp/thread/async

std::atomic
https://en.cppreference.com/w/cpp/atomic/atomic

Definition of Data Races in the C++ Standard
https://timsong-cpp.github.io/cppwp/std20/intro.races

ThreadSanitizer

https://clang.llvm.org/docs/ThreadSanitizer.html

Puzzle 16 An Overloaded Container

an-overloaded-container.cpp

```
#include <initializer_list>
#include <iostream>

struct Container
{
    Container(int, int)
    {
        std::cout << "Two ints\n";
    }
    Container(std::initializer_list<float>)
    {
        std::cout << "std::initializer_list<float>\n
;
    }
};

int main()
{
    Container container1(1, 2);
    Container container2{1, 2};
```

```
}
```

Guess the Output

images/aside-icons/important.png

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Two ints
std::initializer_list<float>
```

Bonus question: what would happen if we removed the `initializer_list` constructor? Would it still compile?

Discussion

The struct `Container` has two constructors, one taking two `int`s and one taking a `std::initializer_list<float>`. Two objects of type `Container` are constructed, both with two `int`s as arguments. But the one difference is that one object is constructed

using parentheses `(1,2)` and the other using curly braces `{1,2}`. Which constructor is called in each case?

For `Container container1(1, 2)`, overload resolution unsurprisingly chooses the constructor taking two `int`s.

For `Container container2{1, 2}`, overload resolution chooses the constructor taking a `std::initialization<float>`, even though we passed it two `int`s! Does this mean `{1,2}` is a `std::initializer_list` then? No, `{1,2}` isn't a `std::initializer_list<int>` but an expression called a *braced-init-list*. A braced-init-list is a pair of curly braces containing zero or more elements. These can occur in several contexts; the most common is for initialization. Here are some examples of braced-init-lists used for initialization:

examples/braced-init-list/braced-init-list.cpp

```
std::array<int, 3> a{1, 2, 3};
int i{2};
int j{};
f({1, 2});
std::map<int, std::string>{{1, "one"}, {2, "two"}};
return {1, 2};
```

As we can see from the initialization of `i` and `j`, even if there's zero or one element and not a "list" of items between the curly-braces, the expression is still called a braced-init-list.

Initialization from a braced-init-list is called *list-initialization*, and the braced-init-list is unfortunately called an *initializer list* in these contexts. An initializer list is not the same as the class template `std::initializer_list<T>`. The names are rather confusing.

So if `{1,2}` isn't a `std::initializer_list`, why does it pick the `std::initializer_list` constructor? Overload resolution has a special case for list-initialization where it happens in two phases:

- First, it does overload resolution for all constructors taking a `std::initializer_list`, treating the braced-init-list as one single argument.
- Then, and only if the first phase didn't succeed, it does overload resolution for the other constructors, treating the elements of the braced-init-list as separate arguments.

So in the case of `Container container2{1, 2}`, overload resolution first tries the `std::initializer_list<float>` constructor. An `int` can be implicitly converted to a `float`, and similarly, an initializer list of `int` can be used to create a `std::initializer_list<float>`. So the `std::initializer_list<float>` constructor is chosen, and the other constructor is never even considered.

Now to the bonus question: what happens if we remove the `std::initializer_list<float>` constructor? For `Container container1(1, 2)`, there's no change since this is not list-initialization. For `Container container2{1, 2}`, overload resolution doesn't find any constructors taking a `std::initializer_list` in the first phase and moves on to the other constructor. The constructor taking two `int`s is used in both cases, and the program still compiles.

std::vector

If you have ever wondered why `std::vector<int> v(1,2)` gives you a vector with a single element, `2`, but `std::vector<int> v{1,2}` gives you a vector with two elements, `1` and `2`, now you know why.

- The first case is not list-initialization, and it picks the constructor taking a `size_type count` and a `T value`.
- The second case is list-initialization, and it picks the constructor taking a `std::initializer_list<T>`.

Further Reading

List-Initialization
https://en.cppreference.com/w/cpp/language/list_initialization

std::initializer_list
https://en.cppreference.com/w/cpp/utility/initializer_list

std::vector Constructors
https://en.cppreference.com/w/cpp/container/vector/vector

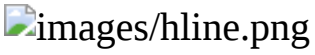
Puzzle 17 A Strong Point

a-strong-point.cpp

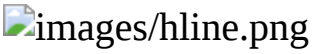
```
#include <iostream>

struct Points
{
Points(int value) : value_(value) {}
int value_;
};

struct Player
{
explicit Player(Points points) : points_(points) {}
Points points_;
};

int main()
{
Player player(3);
std::cout << player.points_.value_;
}
```

Guess the Output

Try to guess what the output is before moving to the next page.



The program displays the following output:



Discussion

The Player struct has a member points_ to keep track of the player's points. Instead of using a fundamental type like int for this, we use a custom type Points. This technique is often called "strong typing" and can make the code easier to read and avoid bugs. For instance, given a function do_something(int player_id, int points, int lives), it can be easy to slip up and pass it player_id, lives, points instead, causing the number of lives to be used for points and vice versa. When you use separate strong types for these values, a mistake like this would cause the program to fail to compile rather than create a bug.

The question in this puzzle is whether we're allowed to construct a Player object directly from an int, like we do in the expression Player player(3), or whether we need to manually create an object of the strong type Points to pass to the Player constructor.

For Player player(3) to work, we need an implicit conversion from int. And only constructors not marked explicit, known as converting constructors, can participate in implicit conversions. Player's constructor is marked explicit and is thus not a converting constructor, so how does this conversion work?

The form Player player(3) is an example of direct-initialization (so is Player player{3} but not, for example, Player player = 3). I like to think that we're directly calling Player's constructor. In direct-initialization, overload resolution is used to pick the best constructor. When we're direct-initializing a Player from an int, overload resolution tries to find an implicit conversion sequence from int to each of Player's constructors. Player has only one constructor, taking a Points, so we need an implicit conversion from int to Points. Points' constructor isn't explicit, so it's a converting constructor and can be used in the implicit conversion sequence. 3 is implicitly converted to Points, and this Points object is passed to Player's constructor.

Player's constructor itself is not involved in the implicit conversion; we're calling it directly, remember. So it makes no difference whether or not it's explicit.

On the other hand, if it was Points' constructor that was explicit, it would no longer be a converting constructor and would not be usable in the implicit conversion sequence. We'd no longer be able to implicitly convert the int 3 to Points, and the program would not compile.

Giving your strong types explicit constructors makes the code slightly more verbose but also more robust. Returning to the do_something example, you would now have to write do_something(PlayerId{42}, Points{3}, Lives{1}) for the program to compile, whereas with converting constructors, you could still slip up and write do_something(42, 1, 3) and the program would still compile. I also find the former more readable.

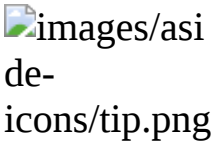
Copy-Initialization vs. Direct-Initialization

What would happen if we instead wrote this:

```
Player player = 3
```

This form is called copy-initialization and differs a bit from direct-initialization. Now we're no longer calling the Player constructor directly; we're instead trying to implicitly convert an int to Player. An implicit conversion sequence can only involve one user-defined conversion, but we would need one from int to Player and then a second one from Player to Points. So Player player = 3 would never compile, regardless of whether any of the constructors are explicit.

Recommendations



- Mark single argument constructors explicit to avoid surprising implicit conversions.
- Use strong types instead of fundamental types to avoid bugs.
- Consider using a strong type library, such as Björn Fahller's strong_type,[13] to reduce boilerplate code when creating strong types.

Further Reading

Direct-Initialization

https://en.cppreference.com/w/cpp/language/direct_initialization

Copy-Initialization

https://en.cppreference.com/w/cpp/language/copy_initialization

C++ Core Guidelines C.46: By Default, Declare Single-Argument Constructors Explicit

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c46-by-default-declare-single-argument-constructors-explicit

Puzzle 18 Moving Out

moving-out.cpp

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string hello{"Hello, World!"};
    std::string other(std::move(hello));
    std::cout << "'" << hello << "'";
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program has unspecified behavior! But it probably outputs the following:

```
''
```

Discussion

A moved-from object should no longer be used, as resources will have been moved out of it, pointers might point to destroyed objects, and so on.

How a particular moved-from object behaves is up to the implementer of that class. The implementer has a lot of freedom here, but at a minimum, the object should be safe to destroy so nothing breaks when, for instance, a moved-from local variable goes out of scope. Unless explicitly documented, you as a user should not assume that a class makes any further guarantees.

The C++ standard provides an additional guarantee for the classes in the standard library, though:

> *Objects of types defined in the C++ standard library may be moved from. (…) Unless otherwise specified, such moved-from objects shall be placed in a valid but unspecified state.*

`std::string` does not specify otherwise, so a moved-from string is placed in a "valid but unspecified state." What does that mean, though?

The C++ standard provides the following definition:

(a) value of an object that is not specified except that the object's invariants are met and operations on the object behave as specified for its type.

So while we don't know what the moved-from string contains, we at least know that `.size()` returns the size of whatever string is left in there, that `.empty()` is `true` if and only if `.size() == 0`, and so on. But still, the actual state of the object is unspecified and we can't be sure what the output will be, so it's better to avoid using moved-from objects altogether.

Implementation of the std::string Move Constructor

One typical way to implement the move constructor for `std::string` is that the new string "steals" the memory from the old string to avoid copying it. It is then important that it's not also possible to access this memory through the old string object, or modification to one string would also affect the other. It's also important that the old string doesn't attempt to free this memory in its destructor. If `std::string` is implemented with a pointer to a string buffer on the heap, one can, for instance, imagine that the move constructor sets the old string's buffer pointer to null. You can use similar techniques to implement move operations for your own classes.

Some implementations will optimize short strings by storing the string inside the `std::string` object itself instead of allocating

Implementation of the std::string Move Constructor

> a buffer on the heap, a technique known as Short String Optimization. In these cases, it's not possible for a move constructor to steal the memory of the old string, and a copy is needed. One reasonable valid state for such a moved-from string would be to still contain the original string.
> I tested this on Windows, Mac, and Linux, both with long and short strings. In all cases they output the empty string `''`. But as always with unspecified behavior, I won't assume this to always be true. I will only assume that the `std::string` invariants are met.

Recommendations

Recommendations

- Do not use moved-from objects (aside from destroying them).
- In your own classes, try to make the move constructors and move assignment operators place the object in a valid state where invariants are still met.
- If you can't do that, at least make sure it's still safe to call the destructor.

Further Reading

std::string Constructors

https://en.cppreference.com/w/cpp/string/basic_string/basic_string

Moved-from State of Library Types

https://timsong-cpp.github.io/cppwp/std20/lib.types.movedfrom

Valid but Unspecified State

https://timsong-cpp.github.io/cppwp/std20/defns.valid

Puzzle 19 A Little Sum Thing

a-little-sum-thing.cpp

```
#include <iostream>
#include <numeric>
#include <vector>

int main()
{
    std::vector<int> v{-2, -3};
    std::cout << std::accumulate(v.cbegin(),
v.cend(), v.size());
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program has implementation-defined behavior! But it outputs a large number, like this one for instance:

```
18446744073709551613
```

Discussion

The size of the vector is `2`. When we add `-2` and `-3` to it, why is the result not `-3`?

The `std::accumulate` function template is defined like so:

```
template<class InputIterator, class T>
constexpr T accumulate(InputIterator first,
InputIterator last, T init);
```

> *Effects: computes its result by initializing the accumulator `acc` with the initial value `init` and then modifies it with `acc = std::move(acc) + *i` (…) for every iterator `i` in the range `[first, last)` in order.*

Let's start by deducing the template parameters. `InputIterator` will be deduced to the iterator type for the vector, which isn't interesting for this puzzle since it's only used for iterating. `T`, on the other hand, will be deduced to the type of `std::vector<int>::size()`, which is key to the puzzle. `std::vector<int>::size()` returns `size_type`, which is not specified in detail by the standard. It's only required to be an unsigned integer type, and as long as that requirement is met, the

implementation can choose whatever representation is natural to use on that particular system.

So both the type of `init` and the return type as well as the accumulator `acc` that's inside `std::accumulate` will be of this unsigned integer type `size_type`. When we add negative numbers to an unsigned type, something is bound to go wrong. But what exactly? And, importantly, is it undefined behavior?

Two error scenarios are worrisome:

- *When converting a value of one integer type to another integer type if the value is out of range for the new type.* This is not as dangerous as it sounds, as the operation doesn't actually overflow but rather wraps around. This is true both for unsigned integers and, as of C++20, also for signed integers. For example, an 8-bit signed integer such as `int8_t` can represent values from -128 to 127. If I try to assign the value 128 to it, which is one too large, it wraps around to -128. When a value wraps around, you're bound to get a wrong result, but at least it's not undefined behavior.
- *When doing arithmetic, if the result is out of range for the result type.* For unsigned integers, we again wrap around. For signed integers, however, we get overflow, which is undefined behavior.

Since we're adding the negative numbers -2 and -3 to our unsigned `acc`, we can at least expect some wraparound, but can we expect overflow and thus undefined behavior? That depends on whether we ever end up adding two signed integers whose sum is out of range for the result type. Let's have a closer look at what happens in `std::accumulate` to see if that's possible.

The algorithm starts by initializing the accumulator `acc` with the size 2. `acc` is of type `T`, which we previously deduced to the unspecified unsigned integer type `size_type`. We then add the signed `int`s `-2` and `-3` to `acc`, one at a time. Finally, we return `acc`. These are all the steps spelled out:

```
size_type acc = 2; // size_type is unsigned
acc = acc + -2;    // Adding an unsigned and a
negative signed number
acc = acc + -3;    // Adding an unsigned and a
negative signed number
return acc;
```

Now, what happens when we add the two signed `int`s -2 and -3 to the unknown unsigned integer type `size_type`? In Puzzle 14, Puzzle 14, *Aristotle's Sum of Parts*, we learned about the usual arithmetic conversions and integer promotions, which also happen here.

Since we don't know the widths of `size_type` and `int`, we need to consider three scenarios:

If size_type Fits in int

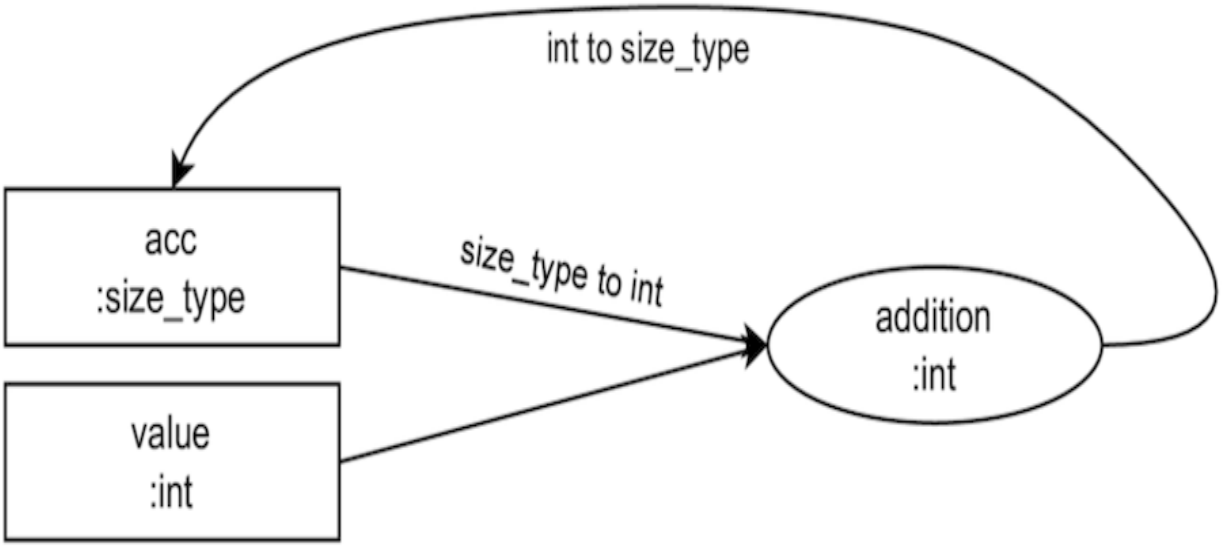
If `int` can represent all of the values of `size_type`, `acc` gets promoted to `int`, and the addition happens with the type `int`. Since `size_type` is unsigned, this can only happen if `size_type` is smaller than `int`. It's very unlikely that an implementation

would choose a `size_type` smaller than `int`, but it would be okay according to the standard. Here's what would happen in this scenario.

First, we add `-2` to `acc`. After promoting `acc` from `size_type` to `int`, the operands are of the same type (`int`), and no further conversion is needed. We add `2` and `-2` and get `0`, which fits in the result type `int`. This `int 0` gets converted back to `size_type` and stored in `acc`.

Next, we add `-3` to `acc`. After promoting `acc` from `size_type` to `int`, we add `0` and `-3` and get `-3`, which again fits in the result type `int`. The `int -3` gets converted back to `size_type`. Since `size_type` is unsigned, `-3` wraps around to `std::numeric_limits<size_type>::max() - 2`, which gets stored in `acc` as the final result and returned. If `size_type` was, for instance, `uint16_t`, the result would be `65533`. This produced a wraparound and a wrong result, but no undefined behavior.

Here's a diagram of the types and conversions involved in the process. `acc` is promoted to `int`, is added to the first value, and the result converts back to `size_type` to get stored in `acc`. Then we go around one more time for the second value.



The `size_type` to `int` promotion is unproblematic since `size_type` fits in `int`. The signed addition has the potential for overflow with undefined behavior if the result doesn't fit in `int` (but that doesn't happen given the input in this puzzle). The `int` to `size_type` conversion has the potential for wraparound if the result is negative (as happens in this puzzle) or if the value is too large (does not happen in this puzzle).

If size_type Does Not Fit in int but Fits in unsigned int

If `int` cannot represent all of the values of `size_type` but `unsigned int` can, `acc` gets promoted to `unsigned int`. This can only happen if `size_type` is the same width as `int` and `unsigned int`. If `size_type` was smaller, it would fit in `int`; if it was wider, it would not fit in `unsigned int`.

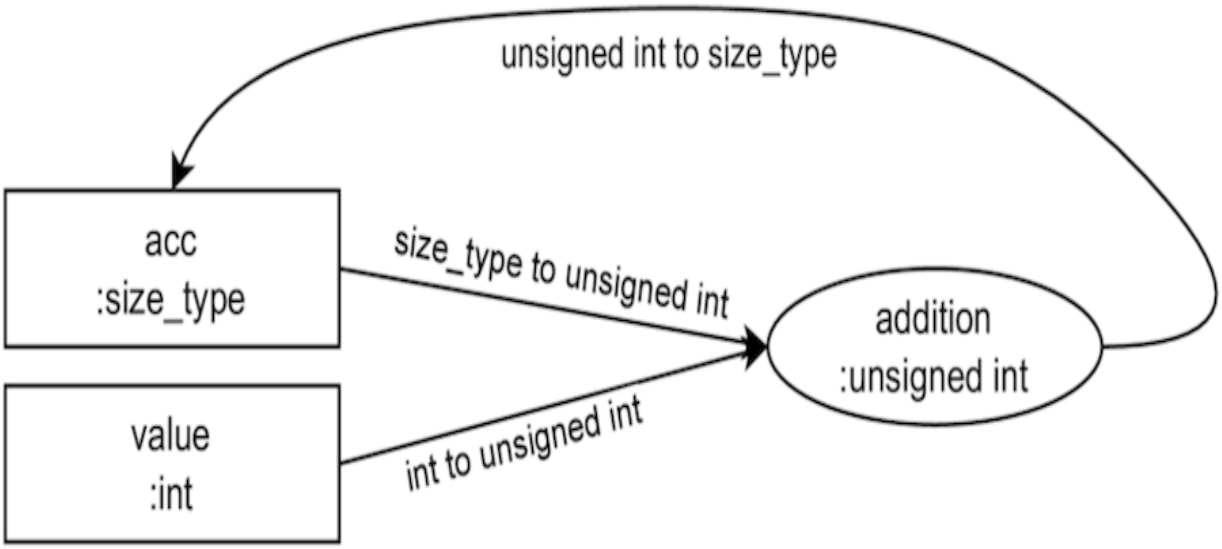
After `acc` has been promoted from `size_type` to `unsigned int`, the operands are still not of the same type, so the usual arithmetic conversions continue. Since `size_type` and `int` have the same width, they also have the same rank, and the `int`

is converted to `unsigned int` (see The Usual Arithmetic Conversions[14] for details). The addition happens with the type `unsigned int`.

First, we add `-2` to `acc`. `-2` is converted to `unsigned int`, and since it is negative, it wraps around to `std::numeric_limits<unsigned int>::max() - 1`. When added to `acc`, which has the value `2`, we wrap around again to `0`.

Next, we add `-3` to `acc`. `-3` is converted to `unsigned int`, and again since it is negative, it wraps around to `std::numeric_limits<unsigned int>::max() - 2`. When added to `acc`, which has the value `0`, the result is `std::numeric_limits<unsigned int>::max() - 2`, which gets stored in `acc` as the final result. On a typical system with 32-bit `int`, the result would be `4294967293`. This produced several wraparounds and a wrong result, but no undefined behavior.

Here's a diagram of the types and conversions involved in the process. `acc` is promoted to `unsigned int` and the first value is converted to `unsigned int`, they are added together, and the result is converted back to `size_type` to get stored in `acc`. Then we go around one more time for the second value.



The `size_type` to `unsigned int` promotion is unproblematic since `size_type` and `unsigned int` are both unsigned and of the same width. The `int` to `unsigned int` conversion has the potential to wrap around if the value is negative (which happens, given the input in this puzzle). The unsigned addition has the potential to wrap around if the result is too large (does not happen in this puzzle). The `unsigned int` to `size_type` conversion is unproblematic since `unsigned int` and `size_type` are both unsigned and of the same width.

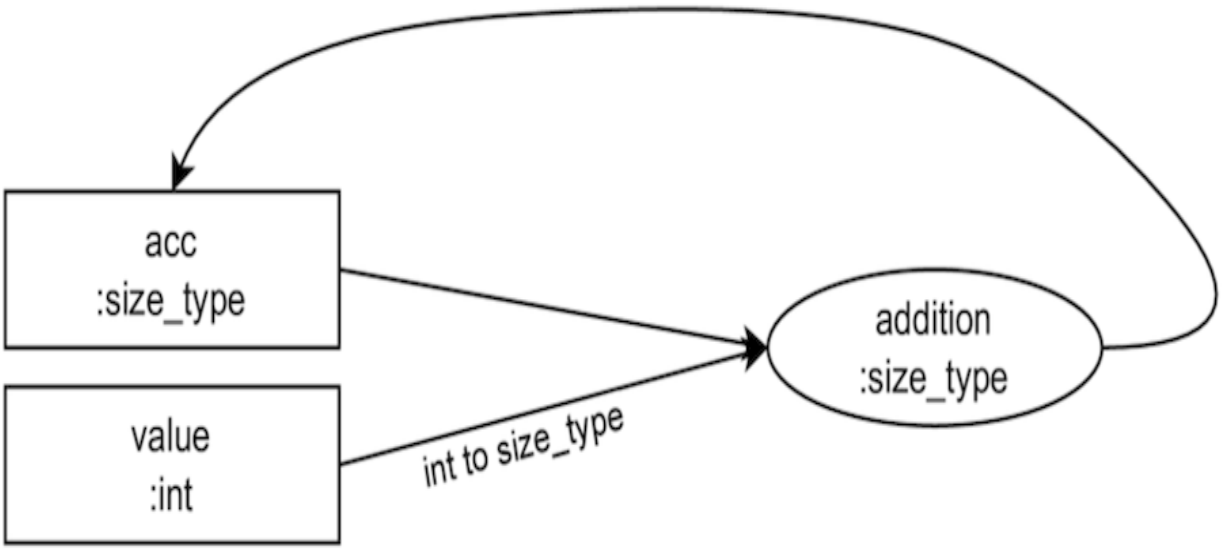
If size_type Does Not Fit in Either int or unsigned int

This is the most common scenario if, for example, `int` is 32 bits and `size_type` is a 64-bit unsigned type. In this case, integral promotion on `size_type` is not performed; we only have the rest of the usual arithmetic conversions to worry about. Since `size_type` is wider than `int`, `size_type` has greater rank, which means that the `int` is converted to the unsigned type `size_type`, and the addition happens with the type `size_type`.

First, we add `-2` to `acc`. `-2` is converted to `size_type` and wraps around to `std::numeric_limits<size_type>::max() - 1`. When added to `acc`, which has the value `2`, we wrap around again to `0`.

Next, we add `-3` to `acc`. `-3` is converted to `size_type` and wraps around to `std::numeric_limits<size_type>::max() - 2`. When added to `acc`, which has the value `0`, the result is `std::numeric_limits<size_type>::max() - 2`, which gets stored in `acc` as the final result. On a typical system with 32-bit `int` and 64-bit `size_type`, the result would be `18446744073709551613`. Again, we saw several wraparounds and a wrong result, but no undefined behavior.

Here's a diagram of the types and conversions involved in the process. The first value is converted to `size_type`, is added to `acc`, and the result gets stored in `acc`. Then we go around one more time for the second value.



The `int` to `size_type` conversion has the potential to wrap around if `value` is negative, which happens given the input in this puzzle. The unsigned addition

has the potential to wrap around if the result is too large (which does not happen in this puzzle).

Are You Still There?

I omitted many details above, especially the exact rules resulting in these particular conversion sequences, and still, the explanation got quite long. If you followed to the end, great, you got the point. If you dropped off at some point though, no worries, as you still got the point! The point being that reasoning about implicit conversions between integers is complicated and best avoided in production code.

What Can We Do?

Unfortunately, none of the major compilers warn about this, even with `-Wall -Wextra` for GCC and Clang, and `/Wall` for MSVC. Some compilers, such as GCC and Clang, have additional options like `-Wsign-conversion`, which are not included in `-Wall` or `-Wextra` and must be enabled individually. Unfortunately, even after enabling them, you get no warnings for this particular puzzle. (You would if the `accumulate` function template was defined in your own code rather than in the standard library, though.)

Clang-tidy is also unable to warn about this, and undefined behavior sanitizer would only trigger if undefined behavior actually happened during the execution of the program, which is not the case.

So, for this particular puzzle, there's not much help to get from tools, and we just have to be aware of the potential problems when mixing integer types and signedness. However, the following is still good advice and can help in similar scenarios:

Recommendations

- Don't mix signed and unsigned types in arithmetic, assignments, or comparisons.
- In addition to `-Wall` and `-Wextra`, manually enable options like `-Wsign-conversion`.
- Use clang-tidy and sanitizers.
- Be aware that warnings and tools will not always find all potential problems.

Further Reading

The Usual Arithmetic Confusions
https://shafik.github.io/c++/2021/12/30/usual_arithmetic_confusions.html

Core Guideline ES.100: Don't Mix Signed and Unsigned Arithmetic
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-mix

Puzzle 20 Monsters on the Move

monsters-on-the-move.cpp

```
#include <iostream>

struct Monster
{
    Monster() = default;
    Monster(const Monster &other)
    {
        std::cout << "Monster copied\n";
    }
    Monster(Monster &&other)
    {
        std::cout << "Monster moved\n";
    }
};

struct Jormungandr : public Monster
{
    Jormungandr() = default;
    Jormungandr(const Jormungandr &other) :
Monster(other)
```

```
    {
        std::cout << "Jormungandr copied\n";
    }
    Jormungandr(Jormungandr &&other) : Monster(other
    {
        std::cout << "Jormungandr moved\n";
    }
};

int main()
{
    Jormungandr jormungandr1;
    Jormungandr
jormungandr2{std::move(jormungandr1)};
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Monster copied
Jormungandr moved
```

Discussion

When we initialize the second `Jormungandr` with a moved-from object, the move constructor gets called for `Jormungandr` but the copy constructor for `Monster`. Why is that?

As we learned in Puzzle 8, Puzzle 8, *Will It Move?*, when we wrap an expression in `std::move`, it turns into an rvalue. An important detail here, though, is that value categories are about expressions, not objects. So nothing happens to the object `jormungandr1` itself; it's just that the *expression* `jormungandr1` is an lvalue, but the expression `std::move(jormungandr1)` is an rvalue. If you ask me, it would be simpler if it was called "l-expression" and "r-expression" instead of "lvalue" and "rvalue."

When we then copy-initialize `jormungandr2` with the expression `Jormungandr jormungandr2{std::move(jormungandr1)}`, overload resolution prefers `Jormungandr`'s move constructor over its copy constructor since the expression `std::move(jormungandr1)` is an rvalue, which can be moved from. And yes, you read that right, this form is always called *copy-initialization* even if the move constructor happens to be called. There's no such thing as move-initialization.

Now let's turn our attention to the implementation of `Jormungandr`'s move constructor. You might already have spotted the bug: we forgot to wrap `other` in a `std::move` like so:

examples/which-constructors/which-constructors.cpp

```
Jormungandr(Jormungandr &&other) :
Monster(std::move(other))
```

The parameter `other` is an rvalue reference. But the *expression* `other`, which we pass to the `Monster` constructor, is an lvalue! Since the expression `other` is an lvalue, we can't use the move constructor, and overload resolution picks the copy constructor instead. However, after wrapping it in `std::move`, the expression `std::move(other)` is now an rvalue, and the move constructor is picked.

The output would then instead be the following:

```
Monster moved
Jormungandr moved
```

Bonus question: what would happen if `Monster` was defined like this instead?

examples/which-constructors/which-constructors.cpp

```
struct Monster
{
    Monster() = default;
```

```
    Monster(const Monster &other) {
        std::cout << "Monster copied\n";
    }
};
```

Since we've explicitly declared a copy constructor, we don't implicitly get a default or move constructor. We have explicitly declared a default constructor, but no move constructor, so `Monster` doesn't have one.

When `Jormungandr`'s move constructor does `: Monster(std::move(other))`, the expression `std::move(other)` is still an rvalue. More specifically, as we saw in Puzzle 9, Puzzle 9, *Counting Copies*, when you wrap an lvalue in `std::move()`, it turns into a type of rvalue called an xvalue. This means the object still has an identity like an lvalue, but we're also allowed to move from it like an rvalue. If `Monster` had a move constructor here (with a `Monster &&` parameter), overload resolution would prefer it over the copy constructor (with the `const Monster &` parameter) when passed an rvalue. But since it doesn't have a move constructor and const lvalue references are allowed to bind to rvalues, the copy constructor is used instead.

We're then back to this:

```
Monster copied
Jormungandr moved
```

Further Reading

Lvalues, Rvalues, Glvalues, Prvalues, Xvalues, Help!
https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/

The Difference Between No Move Constructor and a Deleted Move Constructor
https://blog.knatten.org/2021/10/15/the-difference-between-no-move-constructor-and-a-deleted-move-constructor/

Puzzle 21 Sizing Up Some Characters

sizing-up-some-characters.cpp

```
#include <iostream>

void serialize(char characters[])
{
    std::cout << sizeof(characters) << "\n";
}

int main()
{
    char characters[] = {'a', 'b', 'c'};
    std::cout << sizeof(characters) << "\n";
    std::cout << sizeof(characters) / sizeof
(characters[0]) << "\n";
    serialize(characters);
}
```

Guess the Output

Guess the Output	
images/aside-icons/important.png	Try to guess what the output is before moving to the next page.

The program has implementation-defined behavior! But on a 64-bit CPU it probably outputs the following:

```
3
3
8
```

Discussion

The program defines an array `characters` with three elements. We then print the size of this array in various ways. What does the `sizeof` operator do in each case? Which of them are defined by the standard, and which are implementation-defined? And what do they print on your computer?

First, the program defines an array `char characters[] = {'a', 'b', 'c'}`. We then print the size of this array, using the `sizeof` operator. The `sizeof` operator yields the number

of bytes occupied by its operand, which in this case is `3`, since each `char` is one byte, and there are three of them.

But wait, in Puzzle 14, Puzzle 14, *Aristotle's Sum of Parts*, we learned that the sizes of the fundamental types are not defined by the standard, so how can we be so sure that the answer is 3? A `char` could, after all, be 16 bits wide for all we know. As it turns out, `sizeof(char)` is defined to always return `1`. So the "byte" that the C++ standard talks about isn't necessarily eight bits wide but rather whatever width a `char` is.

Next, we print `sizeof(characters) / sizeof(characters[0])`. This is the age-old idiom used to count the number of elements in an array in C or C++ and is guaranteed to print `3` in this puzzle. This is because `sizeof(characters)` returns `3` (as we just saw), and `sizeof(characters[0])` is equivalent to `sizeof(char)`, which always returns `1`. (As of C++17, we can also count the elements in an array with the new `std::size` function template, which makes the code a bit simpler: `std::cout << std::size(characters)`.)

Finally, we want to serialize our characters, so we pass them to `void serialize(char characters[])`. The serialization function wants to know how large the array is for purposes of serialization and does `sizeof(characters)`, just like we do in `main`. But why does this print `8` (on my computer in particular) and not `3`? It's because the type of the `char characters[]` parameter is actually not "array of `char`" but rather "pointer to `char`"! The standard says the following about the types of function parameters:

> *After determining the type of each parameter, any parameter of type "array of T" or of function type T is adjusted to be "pointer to T".*

So the `sizeof(characters)` in the `serialize` function prints the size of a pointer rather than the size of an array since the type of `characters` is actually "pointer to `char`" at this point. This is the implementation-defined part of the puzzle since the standard doesn't define the size of a pointer. On most modern computers a pointer is 64 bits, which equals 8 bytes, so on your computer you probably see `8` printed.

Now, could we use the seemingly more powerful `std::size` function template here, too, to get the actual number of elements in the array? We can't, since `characters` is a pointer and not an array. `std::size` is only defined for collections and views with a `.size()` member function, and for arrays. `std::size(characters)` would fail to compile rather than giving the wrong result, which is nice.

Recommendations

Recommendations

- Arrays in C++ are very limited, and when you pass them to functions, information like size is lost. Prefer types like `std::vector` or `std::array` instead.
- Enable warnings in your project. GCC and Clang will both warn about the use of `sizeof(characters)` in `serialize`. Here's GCC: `warning: sizeof on array function parameter will return size of 'char *' instead of 'char[]'`. (At the time of writing, I couldn't get MSVC to give me a warning about this.)

Further Reading

sizeof Operator
https://en.cppreference.com/w/cpp/language/sizeof

std::size Function Template
https://en.cppreference.com/w/cpp/iterator/size

Puzzle 22 The Phantom Spaceship

the-phantom-spaceship.cpp

```cpp
#include <iostream>
#include <string>

struct GameObject
{
    GameObject() { std::cout << "Created a " << getType() << "\n"; }
    void render() const { std::cout << "Rendered a " << getType() << "\n"; }
    virtual std::string getType() const { return "GameObject"; }
};

class Spaceship : public GameObject
{
    std::string getType() const override { return "Spaceship"; }
};

void display(const GameObject &gameObject) {
```

```
gameObject.render(); }

int main()
{
    GameObject gameObject;
    Spaceship spaceship;
    display(gameObject);
    display(spaceship);
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
Created a GameObject
Created a GameObject
Rendered a GameObject
Rendered a Spaceship
```

Discussion

In this puzzle, let's say you've attempted to add logging to your game to keep track of the creation and rendering of game objects. You've decided to use the Template Method Design Pattern, which can be a great way to achieve runtime polymorphic behavior in your design. In this pattern, a base class defines common behavior in a concrete function, which internally calls virtual functions that can be overridden in derived classes to fill in the details. For example, the preceding `render` function is a template method that calls the virtual `getType` function to fill in the details of the specific derived type.

This works perfectly for the `render` function, but something goes wrong in the constructor; it prints `Created a GameObject` both when creating a `GameObject` and when creating a `Spaceship`. What exactly is going on, and could this be undefined behavior?

In `main`, when initializing an object of type `Spaceship`, the `GameObject` base class part of the object is initialized first, and then the `Spaceship` part of the object is initialized. During initialization of the `GameObject` part, there's no `Spaceship` part yet. As far as C++ is concerned, this is simply a `GameObject` being created.

So what happens when we call the virtual `getType()` member function in the `Spaceship` constructor, which would normally call down to a `Spaceship` which doesn't even exist yet? Undefined behavior? Luckily, this behavior is well defined. When a virtual function is called from a constructor or destructor, the function called is the one in that particular class and not one overriding it in a

more-derived class. So in the `GameObject` constructor, `GameObject::getType()` is called both when creating a `GameObject` and when creating a `Spaceship`. `Created a GameObject` is displayed in both cases.

In `render`, however, we're dealing with fully initialized objects, and the virtual function works as intended. When we render the `GameObject`, `Rendered a GameObject` is printed, and when we render the `Spaceship`, `Rendered a Spaceship` is printed.

So how do we fix this? Before we look into ways of making the actual type being instantiated available to the `GameObject` constructor, it's worth asking whether it should even be possible to instantiate `GameObject` objects on their own. Does it make sense in the game to be able to instantiate an "abstract" game object? Or does it only make sense to instantiate concrete things like, say, a spaceship, an asteroid, or an astronaut? If we make `GameObject::getType()` a *pure virtual function* rather than giving it a definition, `GameObject` is now an abstract class, which cannot be instantiated on its own:

examples/spaceship/spaceship-pure.cpp

```
virtual std::string getType() const = 0;
```

How does making `GameObject::getType()` pure virtual affect this program though? If you tried it, you may already have noticed that creating a `GameObject gameObject` no longer compiles. You can't instantiate objects of abstract classes, only of concrete derived classes like `Spaceship`. Additionally, creating a `Spaceship spaceship` now results in undefined behavior since the base class `GameObject`

constructor calls a pure virtual function. In practice, you can expect to see either compiler warnings for the call to the pure virtual function, linker errors about a missing `GameObject::getType` definition, or runtime errors like “Pure virtual function called.” But as always with undefined behavior, this varies between platforms and compilers. Is going from the wrong result to undefined behavior an improvement, though? In this case I think so since, given a decent CI setup, you’re likely to discover the problem and fix it.

How do we fix this properly, then? Since there’s no way for the `GameObject` constructor to know which derived type is actually being constructed, we have to somehow provide it with that information. The easiest solution is to pass the information directly to the constructor with a constructor parameter, like so:

examples/spaceship/spaceship-parameter.cpp

```
protected:
  GameObject(std::string_view type) {
      std::cout << "Created a " << type << "\n";
  }
```

This new constructor is protected since it’s only intended for being called from derived classes. We do that in `Spaceship` like so:

examples/spaceship/spaceship-parameter.cpp

```
public:
  Spaceship() : GameObject("Spaceship"){}
```

(Note that the “Spaceship” string is now duplicated between the constructor and the `getType` function and should probably be extracted to a `static constexpr` member of the `Spaceship` class.)

If we still want to be able to instantiate “raw” `GameObject`s, however, we need to add back a default constructor:

examples/spaceship/spaceship-parameter.cpp

```
public:
  GameObject() : GameObject("GameObject"){}
```

This introduces a new problem, though: if you forget to add a user-defined constructor for `Spaceship`, the defaulted `Spaceship` constructor will simply use the `GameObject` default constructor, and we’re back to seeing `GameObject created` when creating a `SpaceShip`—yet another reason to not allow `GameObject`s to be independently constructed.

Finally, when the issue is how to use information from a derived class in a base class without using dynamic polymorphism, no discussion is complete without at least a mention of the Curiously Recurring Template Pattern (CRTP) idiom. While we don’t have space for a detailed discussion here, the idea is to make the base class a template and then use the derived class as a template parameter for its own base class. Now the base class knows, statically even, which derived class is being instantiated simply by looking at its template parameter! Here’s a full example:

examples/spaceship/spaceship-crtp.cpp

```
#include <iostream>

struct GameObject {
    virtual void render() const = 0;
};

template<typename Derived>
struct LoggingGameObject : public GameObject {
    LoggingGameObject() {
        std::cout << "Created a " <<
Derived::typeName << "\n";
    }
    void render() const override {
        std::cout << "Rendered a " <<
Derived::typeName << "\n";
    }
};

struct Spaceship : public
LoggingGameObject<Spaceship> {
    static constexpr auto typeName = "Spaceship";
};

void display(const GameObject &gameObject) {
```

```
gameObject.render(); }

int main() {
    Spaceship spaceship;
    display(spaceship);
}
```

For a more detailed discussion of CRTP, please see Wikipedia.[15]

Recommendations

- Enable warnings. Compilers like GCC and Clang can, for instance, warn about calls to pure virtual functions in constructors and destructors.
- Use several compilers and platforms in CI. Different compilers and linkers give different warnings and errors.

Further Reading

Template Method Design Pattern

https://en.wikipedia.org/wiki/Template_method_pattern

Curiously Recurring Template Pattern (CRTP)

https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Puzzle 23 Off to a Good Start

off-to-a-good-start.cpp

```cpp
#include <iostream>

struct Logger {};
struct Configuration {};

Logger initializeLogger()
{
    std::cout << "Initializing logger\n";
    return Logger{};
}

Configuration readConfiguration()
{
    std::cout << "Reading configuration\n";
    return Configuration{};
}

void startProgram(Logger logger, Configuration
configuration)
{
```

```
    std::cout << "Starting program\n";
}

int main()
{
    startProgram(initializeLogger(),
readConfiguration());
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program has unspecified behavior! But it will display one of two outputs. This is one possible output:

```
Initializing logger
Reading configuration
Starting program
```

And here's the other:

```
Reading configuration
Initializing logger
Starting program
```

Discussion

In this puzzle, we want to start a program that relies on a logger having been set up and a configuration having been read. As long as both of those are done before the program starts, we're good. The order in which the logger is set up and the configuration is read doesn't seem to matter here, but what if it did? What if, for instance, initializing the logger sets up some global logger instance that we use while reading the configuration? Can we rely on the two functions being called in the order listed in the program? Unfortunately, we can't since the evaluation order of function argument expressions is unspecified. This is not only theoretical; when I tested this on Linux, GCC went right-to-left and Clang left-to-right. You can see an online demo at Compiler Explorer.[16]

All is not chaos, however; we do have a few guarantees:

1. Initialization of parameters, including all associated side effects, is guaranteed to have finished before the function starts executing. This guarantees that `initializeLogger` and `readConfiguration` have finished executing before `startProgram` starts executing, and we don't, for instance, risk the

output `Initializing logger` being interleaved with `Starting program`. It sounds obvious, but hey, this is C++, so it's good to check.
2. Initialization of one parameter, including all associated side effects, is guaranteed to have finished before the initialization of another parameter starts. This guarantees that either `initializeLogger` happens entirely before `readConfiguration` or vice versa.
3. There is no undefined behavior here; it's just that the argument expressions can be evaluated in either order.

We know that both `initializeLogger` and `readConfiguration` happen before `startProgram`, just not in which order. Furthermore, we're not even guaranteed that they happen in the same order each time we run the program, even using the same compiler and the same computer! It's unlikely that a compiler would generate code where this is non-deterministic, but it would be okay according to the standard. It's also possible for a compiler to change the ordering in a new version of the compiler if the implementer decides, for instance, that right-to-left is now more efficient than left-to-right.

What can we do about this then? If the ordering is actually important, you can simply call the functions one after the other in the order you desire:

examples/deterministic-order/deterministic-order.cpp

```
int main()
{
    const auto logger = initializeLogger();
    const auto configuration = readConfiguration();
```

```
    startProgram(logger, configuration);
}
```

Note that this is not something compilers can warn about. Without some magical ability to know the semantics of your entire program, there's no way for the compiler to know whether the evaluation order of certain function parameters matters to your program.

Recommendations

- Don't rely on the order of evaluation of function argument expressions.
- Don't assume that expressions are evaluated in a particular order just because they happen to do so in your tests.

Further Reading

Order of Evaluation
https://en.cppreference.com/w/cpp/language/eval_order

Puzzle 24 A Specialized String Theory

a-specialized-string-theory.cpp

```cpp
#include <iostream>

template<typename T>
void serialize(T&) { std::cout << "template\n"; } // 1

template<>
void serialize<>(const std::string&) { std::cout << "specialization\n"; } // 2

void serialize(const std::string&) { std::cout << "normal function\n"; } // 3

int main()
{
    std::string hello_world{"Hello, world!"};
    serialize(hello_world);
    serialize(std::string{"Good bye, world!"});
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
template
normal function
```

Discussion

You might remember our serialization library from Puzzle 2, Puzzle 2, *String Theory*. In the meantime, a C++ expert has come along and decided that everything is better with templates. Whether or not that's true is another discussion, but it certainly makes overload resolution more interesting!

We find three definitions of `serialize` in this puzzle. The first (the one marked "1") is the primary function template for `serialize`. The next (marked "2") is an explicit specialization of this template for `std::string`. You can recognize it as a

specialization by the empty `template<>` tag. Finally, we see a plain old non-template `serialize` function (marked “3”).

Since the name `serialize` is overloaded, when we write a call to that name, overload resolution needs to work out which one to call. As we learned in Puzzle 2, Puzzle 2, *String Theory*, it does that by creating a set of all candidate functions of that name and then selecting the best viable function from that set. It’s clear that the normal, non-template `serialize` function (3) is a candidate, but what happens with the function template? And how is the compiler able to disambiguate between the function template specialization and the normal function, which both have the same signature?

When a function template is involved in overload resolution, the compiler looks at all the *primary templates* of that name. It doesn’t consider specializations at this point, so only the primary template (1) is considered, not the specialization (2). If we had other primary template overloads, these would also be considered, but we only have one in this puzzle. An example of another primary template could, for instance, be this one for pointers:

```
template<typename T>
void serialize(T*) { std::cout << "pointer\n"; }
```

You can recognize this as a primary template by the non-empty `template<typename T>`, just like we saw in the primary template in the puzzle.

Since the primary template (1) is the only primary template in this puzzle, we've now found the function template for `serialize` that should be a candidate for overload resolution. But to be able to to proceed, overload resolution needs to have a concrete function signature with actual types, not just a template taking an arbitrary type `T`. So next, it does type deduction for `serialize` to see what a potential instantiation would look like. This is called "synthesizing a declaration." This synthesized declaration is added to the set of candidate functions (together with the non-template `serialize`), and regular overload resolution proceeds. Let's see how type deduction and overload resolution work for each of the two calls to `serialize`.

First, have a look at the call where we pass an object:

```
serialize(hello_world);
```

The expression `hello_world` is an lvalue of type `std::string`. As we learned in Puzzle 13, Puzzle 13, *A Constant Struggle*, `T` is deduced to `std::string`, and the synthesized declaration has the signature `void serialize(std::string&)`. We now have these two candidates for overload resolution:

```
void serialize(std::string&);       // Synthesized
from the template (1)
void serialize(const std::string&); // The non-
template function (3)
```

Since `hello_world` is non-const, overload resolution prefers the template version of `serialize`, which takes a non-const reference.

Now that overload resolution has decided to use the function template, it's time to find a specialization to use. Should it instantiate and use the primary function template (1) or use the explicit specialization (2)? In this case, that's a simple choice since we need a specialization with the signature `void serialize(std::string&)` but our explicit specialization has the signature `void serialize(const std::string&)`. We have to use the primary function template, and an implicit instantiation is made for `T = std::string`, resulting in `void serialize(std::string&)`. When this instantiation is called, `template` is printed.

Next, let's look at the call where we pass a temporary string:

```
serialize(std::string{"Good bye, world!"});
```

The expression `std::string{"Good bye, world!"}` is a prvalue of type `std::string`, so `T` is deduced to `std::string`, just like for the previous call. We again have these two candidates for overload resolution:

```
void serialize(std::string&);       // Synthesized
from the template (1)
void serialize(const std::string&); // The non-
template function (3)
```

Since `std::string{"Good bye, world!"}` is a prvalue, the non-const reference in the function template can't bind to it. The const reference in the normal function can, and this function is picked by overload resolution. Since the template was not picked, no instantiation takes place, and `normal function` is printed.

Bonus question: what would happen if the function template also took a const reference, like this?

examples/template-vs-function/template-vs-function.cpp

```
template<typename T>
void serialize(const T&) { std::cout << "template\n"
} // 1
```

Now, the synthesized declaration will have the signature `void serialize(const std::string&)`, exactly like the non-template function, and overload resolution needs to pick between these:

```
void serialize(const std::string&); // Synthesized
from the template (1)
void serialize(const std::string&); // The non-
template function (3)
```

If overload resolution finds two viable candidates that are equally good, you normally get a compilation error. So will this even compile?

This is in fact fine. A special clause exists for overload resolution when templates are involved that says if a function F1 is not a function template specialization and F2 is a function template specialization, F1 is a better function than F2. The non-template function is picked for both calls, and the program outputs the following:

```
normal function
normal function
```

Notice that in neither of the preceding cases was the explicit function template specialization even involved in overload resolution—it never is. This can be non-intuitive, so in general it's better to use non-template overloads than explicit specializations. See the first two links under Further Reading for more details and some even worse examples of these kinds of confusions.

Recommendations

- Prefer non-template overloads over explicit specializations.

Further Reading

Why Not Specialize Function Templates?

http://www.gotw.ca/publications/mill17.htm

Full Specialization of Function Templates
https://www.modernescpp.com/index.php/full-specialization-of-function-templates/

C++ Core Guideline T.144: Don't Specialize Function Templates
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#t144-dont-specialize-function-templates

Puzzle 25 Weakly Typed, Strongly Puzzling

weakly-typed-strongly-puzzling.cpp

```
#include <iostream>

int main()
{
    std::cout << +!!"";
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program displays the following output:

```
1
```

Discussion

Wait, what? Is this legal C++? What does it even mean, and why does it print `1`?

While C++ is a statically typed language, it's not particularly strongly typed. As we've seen throughout the book, C++ is, in fact, very eager to implicitly convert between types without you explicitly telling it to. For instance, in Puzzle 14, Puzzle 14, *Aristotle's Sum of Parts*, we saw how the sum of two `char`s is not a `char`; in Puzzle 17, Puzzle 17, *A Strong Point*, we saw how you should declare your constructors `explicit` to avoid them taking part in implicit conversions; and in Puzzle 19, Puzzle 19, *A Little Sum Thing*, we saw how implicit conversions between arithmetic types can lead to unexpected results.

This eagerness is something we inherited back in the day when C++ was designed as an extension of C. If we did it over today, this abundance of implicit conversions would be high on the list of things we'd want to change.

So which implicit conversions are in play here, what is the final type of this expression, and why is the result `1`?

Let's start to the right, with the expression `""`. This is a string literal, and as we discussed in Puzzle 2, Puzzle 2, *String Theory*, the type of a string literal is "array of `n const char`," with `n` being `1` in this case (for the implicit `\0` terminator).

Next is the `!` operator, which negates its operand. But how do you negate an array of characters? That doesn't make sense, does it? Indeed it doesn't, but the

operand to the `!` operator is contextually converted to bool before being negated. Converting an array to bool doesn't make sense either, but as we saw in Puzzle 2, Puzzle 2, *String Theory*, arrays can be implicitly converted to pointers, and pointers can again be converted to bool. When a pointer is converted to `bool`, it converts to `false` if it's a null-pointer and to `true` otherwise. Since the pointer in this case is non-null (it points to our string literal), it converts to `true`. Finally, the `!` operator negates this, so we arrive at `false`. One could argue that this entire conversion sequence doesn't really make sense either, and I'd agree, but at least we can understand how each step works and how they combine to arrive at this result.

Next up is another `!`, which this time simply negates our `bool false`, giving us the `bool true`.

Finally, there's the unary `+`. The operand to unary `+` goes through integral promotions, which for `bool` means converting `false` to `0` and `true` to `1`. The operator itself doesn't do anything; its result is the same as its operand (after promotion). So our `bool true` finally turns into the `int 1`, which gets printed.

Whether or not the compiler should warn about any of this is another discussion, but none of the major ones do, even with `-Wall -Wextra` for GCC and Clang, and `/Wall` for MSVC. Your compiler might be able to provide some warnings, though, if you turn those warnings on explicitly. For instance, Clang will warn about the string-to-bool conversion if given the `-Wstring-conversion` option (which is not included in `-Wall` or `-Wextra`). Check your compiler's documentation for the list of warnings it supports.

Recommendations

- Turn on as many warnings as you can.
- Be aware that not all warnings are included in the maximum warning levels for each compiler.
- Don’t write code like this in real life!

Further Reading

CppReference on Implicit Conversions

https://en.cppreference.com/w/cpp/language/implicit_conversion

Implicit Conversion Sequences in the Standard

https://timsong-cpp.github.io/cppwp/std20/over.best.ics

FOOTNOTES

[6] https://en.cppreference.com/w/cpp/language/initialization

[7] https://timsong-cpp.github.io/cppwp/std20/basic.start

[8] https://en.cppreference.com/w/cpp/language/siof

[9] https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/

[10] https://itanium-cxx-abi.github.io/cxx-abi/abi.html

[11] https://blog.knatten.org/2011/08/26/dont-be-afraid-of-returning-by-value-know-the-return-value-optimization/

[12] https://timsong-cpp.github.io/cppwp/std20/stmt.ranged

[13] https://github.com/rollbear/strong_type

[14] https://timsong-cpp.github.io/cppwp/std20/expr.arith.conv

[15] https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

[16] https://www.godbolt.org/z/P43cETdWd

Thank you!
We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head on over to https://pragprog.com right now, and use the coupon code BUYANOTHER2024 to save 30% on your next ebook. Offer is void where prohibited or restricted. This offer does not apply to any edition of *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With up to a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit https://pragprog.com/become-an-author/ today to learn more and to get started.

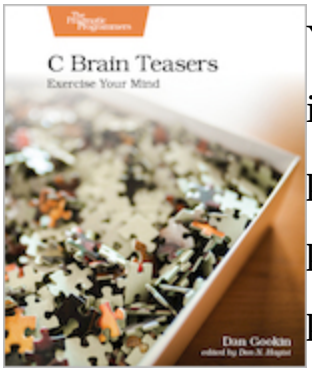
Thank you for your continued support. We hope to hear from you again soon!

The Pragmatic Bookshelf

You May Be Interested In…

Select a cover for more information

C Brain Teasers



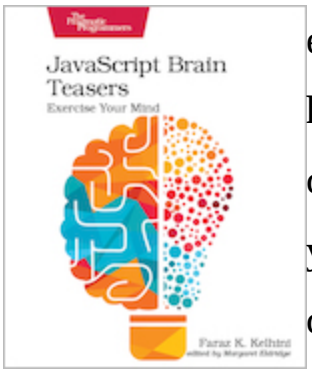
You thought you knew C, but can you solve 25 puzzles in this popular programming language? Noted C programmer and author Dan Gookin provides a series of pointed questions, puzzles, and problems to keep your C programming skills sharp. Each one will provide insight into various aspects of handling strings, numeric operations, and other activities, giving you techniques to take the best advantage of all C has to offer. Challenge yourself, and get to know some powerful tricks and details for writing better, faster, more accurate C code.

Dan Gookin

(118 pages) ISBN: 9798888650486 $32.95

JavaScript Brain Teasers

If you've ever felt the exhilaration of cracking a thorny problem or the joy of witnessing elegant code that challenges your mind, then this book is tailor-made for you. Prepare yourself to dive into a diverse array of mind-bending JavaScript brain teasers. Each puzzle is more than just a code

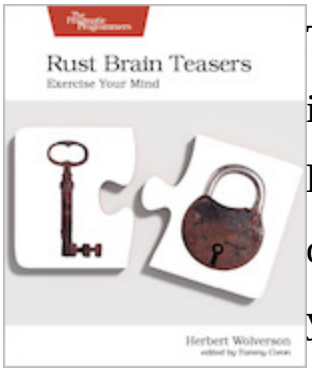


exercise; it’s a gateway to unlocking new insights and honing your abilities. As you work through the challenges, you’ll learn to think pragmatically, optimize your code for efficiency, and avoid hidden pitfalls that can catch you off guard. What’s more—the puzzle format will help you remember what you’ve learned!

Faraz K. Kelhini

(128 pages) ISBN: 9798888650523 $32.95

Rust Brain Teasers

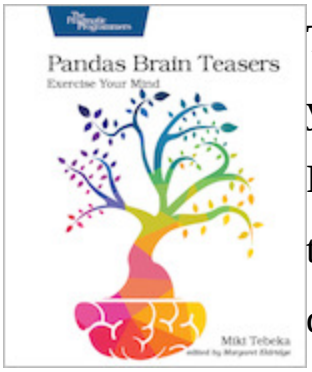


The Rust programming language is consistent and does its best to avoid surprising the programmer. Like all languages, though, Rust still has its quirks. But these quirks present a teaching opportunity. In this book, you’ll work through a series of brain teasers that will challenge your understanding of Rust. By understanding the gaps in your knowledge, you can become better at what you do and avoid mistakes. Many of the teasers in this book come from the author’s own experience creating software. Others derive from commonly asked questions in the Rust community. Regardless of their origin, these brain teasers are fun, and let’s face it: who doesn’t love a good puzzle, right?

Herbert Wolverson

(138 pages) ISBN: 9781680509175 $18.95

Pandas Brain Teasers



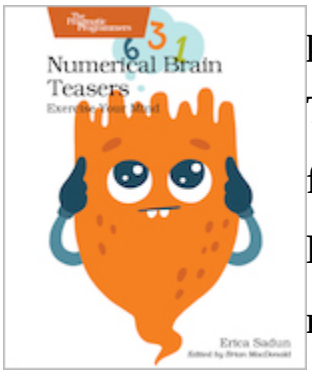
This book contains 25 short programs that will challenge your understanding of Pandas. Like any big project, the Pandas developers had to make some design decisions that at times seem surprising. This book uses those quirks as a teaching opportunity. By understanding the gaps in your knowledge, you'll become better at what you do. Some of the teasers are from the author's experience shipping bugs to production, and some from others doing the same. Teasers and puzzles are fun, and learning how to solve them can teach you to avoid programming mistakes and maybe even impress your colleagues and future employers.

Miki Tebeka

(110 pages) ISBN: 9781680509014 $18.95

Numerical Brain Teasers

Challenge your brain with math! Using nothing more than basic arithmetic and logic, you'll be thrilled as answers slot into place. Whether purely for fun or to test your knowledge, you'll sharpen your problem-solving skills and flex your mental muscles. All you need is logical thought, a little

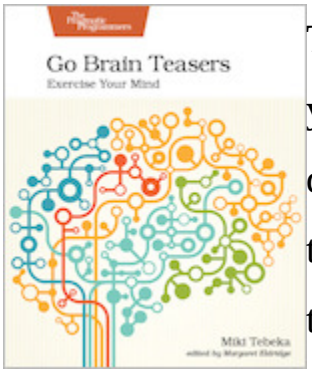


patience, and a clear mind. There are no gotchas here. These puzzles are the perfect introduction to or refresher for math concepts you may have only just learned or long since forgotten. Get ready to have more fun with numbers than you've ever had before.

Erica Sadun

(186 pages) ISBN: 9781680509748 $18.95

Go Brain Teasers

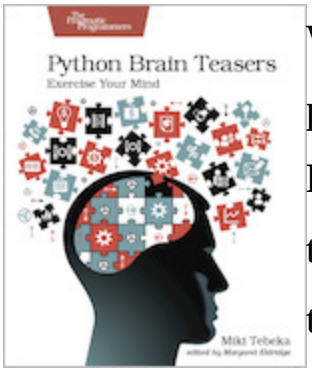


This book contains 25 short programs that will challenge your understanding of Go. Like any big project, the Go developers had to make some design decisions that at times seem surprising. This book uses those quirks as a teaching opportunity. By understanding the gaps in your knowledge, you'll become better at what you do. Some of the teasers are from the author's experience shipping bugs to production, and some from others doing the same. Teasers and puzzles are fun, and learning how to solve them can teach you to avoid programming mistakes and maybe even impress your colleagues and future employers.

Miki Tebeka

(110 pages) ISBN: 9781680508994 $18.95

Python Brain Teasers

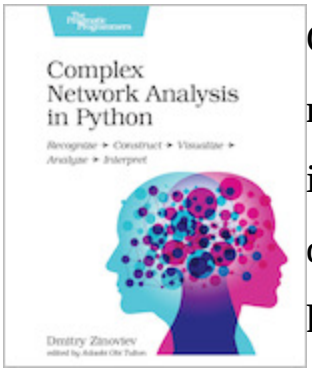


We geeks love puzzles and solving them. The Python programming language is a simple one, but like all other languages it has quirks. This book uses those quirks as teaching opportunities via 30 simple Python programs that challenge your understanding of Python. The teasers will help you avoid mistakes, see gaps in your knowledge, and become better at what you do. Use these teasers to impress your co-workers or just to pass the time in those boring meetings. Teasers are fun!

Miki Tebeka

(116 pages) ISBN: 9781680509007 $18.95

Complex Network Analysis in Python



Construct, analyze, and visualize networks with networkx, a Python language module. Network analysis is a powerful tool you can apply to a multitude of datasets and situations. Discover how to work with all kinds of networks, including social, product, temporal, spatial, and semantic networks. Convert almost any real-world data into a complex network—such as recommendations on co-using cosmetic products, muddy hedge fund connections, and online friendships.

Analyze and visualize the network, and make business decisions based on your analysis. If you’re a curious Python programmer, a data scientist, or a CNA specialist interested in mechanizing mundane tasks, you’ll increase your productivity exponentially.

Dmitry Zinoviev

(260 pages) ISBN: 9781680502695 $35.95

Modern C++ Programming with Test-Driven Development

If you program in == C++ == you’ve been neglected. Test-driven development (TDD) is a modern software development practice that can dramatically reduce the number of defects in systems, produce more maintainable code, and give you the confidence to change your software to meet changing needs. But == C++ == programmers have been ignored by those promoting TDD;—until now. In this book, Jeff Langr gives you hands-on lessons in the challenges and rewards of doing TDD in == C++. ==

Jeff Langr

(366 pages) ISBN: 9781937785482 $40

Test-Driven Development for Embedded C

Still chasing bugs and watching your code deteriorate? Think TDD is only for desktop or web apps? It’s not: TDD is for you, the embedded C programmer. TDD helps you prevent defects and build software with a long useful life. This is the first book to teach the hows and whys of TDD for C programmers.

James W. Grenning

(354 pages) ISBN: 9781934356623 $34.95